



Portable Object Oriented Language

Tutorial, Part 3: Libraries

Revision level: see [Revision index](#)

©

SIEMENS VDO

A u t o m o t i v e

VDO-Straße 1
12345 Babenhausen
Germany



Büro für Datentechnik GmbH
D-35418 Buseck
Germany

1 Contents

1	CONTENTS	2
2	REVISION INDEX.....	5
3	HOW TO WORK WITH THIS TUTORIAL	7
4	ARITHMETICAL FUNCTIONS	8
4.1	Trigonometrical functions.....	9
4.2	Exponential and logarithm functions.....	16
4.3	Power functions	19
4.4	Transfer functions.....	21
4.5	Miscellaneous functions	24
5	STRING AND CHARACTER FUNCTIONS.....	28
5.1	Character functions	29
5.2	Convert ByteString to String.....	32
5.3	General String and ByteString functions(general)(general).....	34
5.4	Functions to convert strings into values.....	53
5.5	Converting charaters to strings	60
5.6	Converting values to strings.....	62
5.7	Special ByteString functions	80
5.8	Struktur QWord	83
5.9	QWord functions	84
6	FILE I/O FUNCTIONS	90
6.1	Introduction	90
6.2	Binary files	92
6.3	Text files.....	111
7	POOL BASE CLASSES.....	129
7.1	The base class toRoot.....	129
7.2	Object toEMRoot.....	136
8	EVENT HANDLING	140
8.1	Introduction	140
8.2	Event objects.....	142
8.2.1	Object toEvent	142
8.2.2	Object toTEvent	148
8.2.3	Object toNWEvent	154
8.3	Event functions	156

9	MONITOR AND KEYBOARD FUNCTIONS	164
9.1	Introduction	164
9.2	Descriptions	165
9.3	Keyboard codes	187
10	MENU FUNCTIONS	190
11	THREAD AND TASK FUNCTIONS	202
11.1	Threads	202
11.2	Tasks	216
12	SYNCHRONISATION OBJECTS (MUTEX)	223
12.1	Introduction	223
12.2	Mutex objects	224
13	ERROR FUNCTIONS	230
13.1	Functions	231
13.2	Error codes	237
14	SYSTEM FUNCTIONS	243
14.1	General system functions	243
14.2	Date and time functions	256
15	GENERAL LIBRARY FUNCTIONS	261
16	COMPILER INTERNAL STANDARD FUNCTIONS	282
16.1	General functions	283
16.2	Dynamic memory management	292
16.3	Variable parameter lists and untyped parameters	299
16.3.1	Functions and procedures for variable parameter lists	300
16.3.2	Functions and procedures for untyped parameters	312
17	MISCELLANEOUS FUNCTIONS	315
17.1	Val function	315
17.2	Memory functions	317
17.3	Relationship of objects	326
17.4	Converting into hexadecimal format	328
17.5	Random number function	330
17.6	Conversion using conversation vectors	332
18	CONSTANTS AND MASKS FROM POOL.PLI	334
18.1	Standard constants (internal to the compiler)	334
18.2	System and Commander constants	334
18.3	General constants and masks	334

18.4	Constants for variable parameter lists and untyped parameters	337
18.5	File constants and masks	339
18.6	Control character constants	340
18.7	Constants for library functions	342
18.8	Bitmasks for tstType.bFlags	342
18.9	Bitmasks for tstField.wFlags	342
18.10	Masks and values for tstOptions.bFlags	343
19	SPECIAL DATA TYPES AND STRUCTURES IN POOL.PLI	344
19.1	File modes.....	344
19.2	Data type QWord	345
19.3	Types for general library functions.....	345
19.4	Structure to adminstrate errors of lib and NoWait functions	346
19.5	Types for system functions	347
	Commander variables.....	348
19.6	Alternative data type denominations	348
20	RECORD TYPES.....	351
20.1	Record for keyboard input	351
20.2	Record for system properties	351
20.3	Records for untyped parameters and variable parameter lists.....	352
20.4	Record for files (Filehandle).....	355
20.5	Records for time and date functions	355
20.6	Miscellaneous records	357
21	SHARED MEMORY FUNCTIONS (SHM.PLI)	360
21.1	Types and structures in shm.pli.....	361
21.2	Shared memory functions.....	362
	ATTACHMENT.....	376
	A1 Bibliography.....	376
	A2 INDEX.....	377

2 Revision index

Date	Author	Rev.	Ref.	Type	Description
2003-05-23	Harald Ebert	0.91.21	div.	cont.	Spelling corrections
2003-05-23	Harald Ebert	0.91.20	div.	cont.	Translated into English
2002-10-29	Uwe Kühn	0.91.10	div.	cont. auth. auth.	Content was revised Text was edited Formatting, form templates
2002-10-16	Thomas Locker	0.91.00	-	-	Initial Revision

Acronyms:**AIDA**

Automotive and Industrial Diagnostic Assistance. System that is used to implement computer supported diagnosis of control modules and bus systems.

BSK

The manufacturer of the AIDA-Systems

EOF

End of File. Indicates the end of the file.

OOP

Object oriented programming (see tutorial part 2).

POOL

Portable Object Oriented Language. Object oriented programming language by BSK. POOL is used for programming in the AIDA system.

SHM

Shared-Memory. Memory block that can be accessed by multiple tasks and/or processes which is used for inter process communication.

SMK

Software Method Kit. Description of the programming guidelines by SiemensVDO.

3 How to work with this tutorial

The third part of the tutorial describes the standard POOL libraries. This part is organized as reference so we do not recommend to read this part from cover to cover. Instead you should just get an overview of the functions and refer to it when you actually need it. A thorough knowledge of the POOL language is required to comprehend the provided examples.

This reference contains short examples that show how to use these functions, contrary to most other references. The examples are kept simple in order to focus on the application of the functions. That is, error handling is omitted resp. reduced to printing an error code and exiting the function or application. Please notice that appropriate error handling is dependent of the application and your responsibility. For more complex examples please refer to the folder in the AIDA installation directory "bsk\aida\examples".

Chapters on more complex topics like file input/output or multithreading provide a short introduction into the subject to explain basic concepts. The subsequent description of functions consists of the function prototype (function name, parameters and return type), an explanation of the function, the parameters, the return value, important conditions, and any other specialities. We provide different views to the relationship of the functions in order to support comprehension. Most descriptions are followed by an example to show possible applications of the functions. Complex topics like multithreading provide complete programming examples to show the relationship of different functions. As stated above, these programs are kept as simple as possible.

Constants, masks, predefined data types and records are summarized in chapters of their own, so you can easily look them up when needed. They are ordered thematically as good as possible. Their usage is explained in detail all over the document scattered in the sections of the functions using them. So the chapters provide only a short explanation taken from the library pool.pli.

In case you can not find the description of a function or predefined data type, please use the index at the end of the document or full text search when you are using a computer. The tutorial is also available as PDF file.

In spite of thorough preparation of the document we can not completely exclude errors. Please notice that neither the author nor BSK GmbH provide any warranties as to the suitability, accuracy, and fitness for a particular purpose. You use the examples at your own risk.

4 Arithmetical functions

Taken from the pool.pli library.

The standard POOL library implements a set of mathematical functions like trigonometrical, exponential, logarithmic, and power functions

Basic mathematical operations (+, -, *, /) and their precedence are covered in tutorial part one.

Please take care to avoid errors resulting out of exceeding data type ranges when doing mathematical calculations.

4.1 Trigonometrical functions

Declaration

```
function ArcCos (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose arc cosine is returned.

Range: $x \in [-1,1]$

Return value

Real64.

Arc cosine of the specified parameter in radians.

Description

The function returns the arc cosine of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value: Real64;
  r64Res:   Real64;
begin
  r64Value := -1;
  r64Res   := ArcCos(r64Value); {calculate arc cosine}
  Writeln(r64Res);           {print the result}
end;
```

ArcSin**Declaration**

```
function ArcSin (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose arc sine is returned (in radians).

Range: $x \in [-1, 1]$

Return value

Real64.

Arc sine of the specified parameter in radians.

Description

The function returns the arc sine of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value: Real64;
  r64Res: Real64;
begin
  r64Value := -1;
  r64Res := ArcSin(r64Value); {calculate arc sine}
  Writeln(r64Res); {print result}
end;
```

ArcTan

Declaration

```
function ArcTan (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose arc tangent is returned (in radians).

Range: $x \in [-\pi/2, \pi/2]$

Return value

Real64.

Arc tangent of the specified parameter in radians.

Description

The function returns the arc tangent of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value: Real64;
  r64Res:   Real64;
begin
  r64Value := -pi/2;
  r64Res   := ArcTan(r64Value); {calculate arc tangent}
  Writeln(r64Res);             {print result}
end;
```

ArcTan2**Declaration**

```
function ArcTan2 (r1,r2: Real64): Real64;
```

Parameter

r1.

Real64. X value in radians.

Range: $x \in \mathbb{R} \setminus 0$

r2.

Real64. Y value in radians.

Range: $x \in \mathbb{R}$

Return value

Real64.

Arc tangent of Y/X the specified vector parameter in radians.

Description

The function calculates the arc tangent of Y/X. ArcTan2 is defined for every point other than the origin (division by 0).

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Value2: Real64;
  r64Res:    Real64;

begin
  r64Value1 := pi/3;
  r64Value2 := pi;
  r64Res    := ArcTan2 (r64Value1,r64Value2); {calculate arc tangent2}
  Writeln(r64Res);                          {print result}
end;
```

Cos**Declaration**

```
function Cos (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value (in radians), whose cosine is returned.

Return value

Real64.

Cosine of the specified parameter.

Description

The function returns the cosine of the specified parameter.

Remarks

The constant pi uses a limited number of decimal places. Therefore Cos(pi/2) is not exactly 1.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := pi;
  r64Res    := Cos(r64Value1); {calculate cosine}
  Writeln(r64Res);           {print result}
end;
```

Sin**Declaration**

```
function Sin (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value (in radians), whose sine is returned.

Return value

Real64.

Sine of the specified parameter.

Description

The function returns the sine of the specified parameter.

Remarks

The constant pi uses a limited number of decimal places. Therefore Sin(pi/2) is not exactly 0.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := pi/2;
  r64Res     := Sin(r64Value1); {calculate sine}
  Writeln(r64Res);             {print result}
end;
```

Tan

Declaration

```
function Tan (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value (in radians), whose tangent is returned.

Return value

Real64.

Tangent of the specified parameter.

Description

The function returns the tangent of the specified parameter.

Remarks

The constant pi uses a limited number of decimal places. Therefore Tan(pi) is not exactly 0.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := pi/2;
  r64Res    := Tan(r64Value1); {calculate tangent}
  Writeln(r64Res);           {print result}
end;
```

4.2 Exponential and logarithm functions

Declaration

```
function Exp (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose exponential value is returned.

Return value

Real64.

The function returns the exponential value of the specified parameter r64.

Description

The function returns e to the power of r64.

Remarks

e (euler's number) is the base of the natural logarithm.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 2;
  r64Res    := Exp(r64Value1); {calculate the exponential value}
  Writeln(r64Res);           {print result}
end;
```

Ln**Declaration**

```
function Ln (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numerical value, whose natural logarithm is returned.

Return value

Real64.

Natural logarithm of the specified parameter r64.

Description

The function returns the natural logarithm.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 2;
  r64Res    := Ln(r64Value1); {calculate natural logarithm}
  Writeln(r64Res);          {print result}
end;
```

Log**Declaration**

```
function Log (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose common logarithm (base is 10) is returned.

Return value

Real64.

Natural logarithm of the specified parameter r64.

Description

The function returns the common logarithm.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 2;
  r64Res    := Log(r64Value1); {calculate common logarithm}
  Writeln(r64Res);           {print result}
end;
```

4.3 Power functions

Declaration

```
function Sqr (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, that is to be raised to the power of two.

Return value

Real64.

Specified parameter r64 raised to the power of two.

Description

The function return the specified parameter raised to the power of two, i. e. the parameter multiplied with itself.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 4;
  r64Res    := Sqr(r64Value1); {calculate r64Value1 raised to the power of 2}
  Writeln(r64Res);           {print result: r64Res = 16}
end;
```

Sqrt**Declaration**

```
function Sqrt (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose square root is returned.

Return value

Real64.

Square root of the specified parameter r64.

Description

The function returns the square root of the specified parameter r64.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 16;
  r64Res    := Sqrt(r64Value1); {calculate the square root}
  Writeln(r64Res);             {print result: r64Res = 4}
end;
```

4.4 Transfer functions

Declaration

```
function Round (r64: Real64): Int32;  
function i32Round (r64: Real64): Int32;  
function dwRound (r64: Real64): DWord;
```

Parameter

r64.

Real64. A numeric value, that is to be rounded and casted to Int32 or DWord.

Return value

Int32 resp. DWord.

Rounded value.

Description

The function rounds to the next integer. With *.5 is rounded to the next bigger absolute integer (+1.5 → +2, -1.5 → -2). Depending on the function the return value is of type Int32 (Round, i32Round) or DWord (dwRound).

Remarks

In case the parameter exceeds the range, the next possible value is returned.

The functions Round and i32Round only differ in the function name.

To get a return value of type r64 use r64Round.

Example

```
procedure vMain;  
var  
    i32Res: Int32;  
  
begin  
    i32Res := Round(2.5);  
    Writeln(i32Res);           {print result:  3}  
    i32Res := Round(-2.5);  
    Writeln(i32Res);          {print result: -3}  
    i32Res := Round(2.4599);  
    Writeln(i32Res);          {print result:  2}
```

Round

```
{Return DWord:}  
  Writeln(dwRound(-2));    {print result:  0}  
end;
```

Trunc

Declaration

```
function Trunc (r64: Real64): Int32;  
function i32Trunc (r64: Real64): Int32;  
function dwTrunc (r64: Real64): DWord;
```

Parameter

r64.

Real64. A numeric value, that is to be rounded and casted to Int32 or DWord.

Return value

Int32 or DWord.

Rounded value.

Description

The function truncates the decimal places, i. e. positiv numeric values are rounded to the next smaller integers and negativ numeric values always return 0.

Remarks

In case the parameter exceeds the range, the next possible value is returned.

The functions Trunc and i32Trunc only differ in the function name.

To get a return value of type r64 use r64Trunc.

Example

```
procedure vMain;  
var  
  i32Res: Int32;  
begin  
  i32Res := Trunc(2.5);  
  Writeln(i32Res);           {print result:  2}  
  i32Res := Trunc(-2.5);  
  Writeln(i32Res);          {print result: -2}  
  i32Res := Trunc(2.4599);  
  Writeln(i32Res);          {print result:  2}  
  {Return DWord:}  
  Writeln(dwTrunc(-2));     {print result:  0}  
end;
```

4.5 Miscellaneous functions

Declaration

```
function Int (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose integer part is returned.

Return value

Real64.

Integer part of the specified parameter.

Description

The function returns the integer part of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 16.7234;
  r64Res    := Int(r64Value1); {calculate integer part}
  Writeln(r64Res);           {print result: r64Res = 16}
end;
```

r64Trunc**Declaration**

```
function r64Trunc (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose integer part is returned.

Return value

Real64.

Integer part of the specified parameter.

Description

The function returns the integer part as type r64 of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 16.1234;
  r64Res    := r64Trunc(r64Value1); {calculate integer part}
  Writeln(r64Res);                 {print result: r64Res = 16}
end;
```

Frac**Declaration**

```
function Frac (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, whose fractional part is returned.

Return value

Real64.

Fractional part of the specified parameter.

Description

The function returns the fractional part as type r64 of the specified parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r64Value1: Real64;
  r64Res:    Real64;
begin
  r64Value1 := 16.7234;
  r64Res    := Frac(r64Value1); {calculate fractional part}
  Writeln(r64Res);             {print result: r64Res = 0.7234}
end;
```

r64Round**Declaration**

```
function r64Round (r64: Real64): Real64;
```

Parameter

r64.

Real64. A numeric value, that is to be rounded.

Return value

Real64.

Rounded value of the specified parameter.

Description

The function rounds to the next integer. With *.5 is rounded to the next bigger absolute integer (+1.5→+2, -1.5 → -2) (see also the example).

Remarks

None.

Example

```
procedure vMain;
var
  r64Res: Real64;
begin
  r64Res := r64Round(12.499); {round value}
  Writeln(r64Res);           {print result: r64Res = 12}

  Writeln(r64Round(12.501)); {print result: 13}
  Writeln(r64Round(-12.499)); {print result: -13}
  Writeln(r64Round(-12.501)); {print result: -13}
  Writeln(r64Round(12.500)); {print result: 13}
  Writeln(r64Round(-12.500)); {print result: -13}

end;
```

5 String and character functions

Taken from the pool.pli library.

For a description of the data types `String` (`CharString`), `ByteString`, and `Char` please see the first part of the tutorial. Due to the great number of string functions we recommend that you only obtain information and learn the functional scope at first. You can come back and delve into operating principle and use of functions in details when it becomes necessary. The description differentiates between the terms `String` or `ByteString` and the data type `String` or `ByteString`; the same distinction is made for the data type `Byte` and the term `byte` resp. `bytes`. When describing functions, which are equally working with both string types the term `sting` is used in order to keep things simple. Since these functions are marked differently (`BSTR`), there is no danger of getting them mixed up.

The pool.pli standard library contains a lot of functions and procedures that can be used to work with and manipulate `String` and `Char` variables.

The functional scope includes functions that are used to convert strings into values and vice versa, and also functions that can be used to compare strings, as well as functions that are used to copy, decompose and compose strings.

Additionally it is also possible with the help of special functions for instance to replace individual characters and influence the writing of uppercase and lowercase letters. Thus, strings can be composed according to your requirements.

Since there are too many functions, this introduction can not describe all of them; therefore, we recommend that you try to get a general idea of the available functions with the help of the following section .

Functions and procedures that are marked with the key word `bstr` can work with both strings (`String`) and byte strings (`ByteString`).

5.1 Character functions

Declaration

```
function ChLower(c: Char): Char;
```

Parameter

c.

Char. A letter that is to be converted into a lower case letter.

Rückgabwert

Char.

Lower case letter or the passed parameter.

Description

The function converts an upper case letter into a lower case letter. In case you pass a lower case letter or a special character no conversion takes place.

Remarks

None.

Example

```
procedure vMain;
var
  cTest: Char;
begin
  cTest := "K";
  Writeln(cTest);           {print: K}
  cTest := ChLower(cTest); {convert to lower case letter}
  Writeln(cTest);           {print: k}
end;
```

ChUpper

Declaration

```
function ChUpper(c: Char): Char;
```

Parameter

c.

Char. A letter that is to be converted into an upper case letter.

Rückgabwert

Char.

Upper case letter or the passed parameter.

Description

The function converts a lower case letter into an upper case letter. In case you pass an upper case letter or a special character no conversion takes place.

Remarks

None.

Example

```
procedure vMain;
var
  cTest: Char;
begin
  cTest := "k";
  Writeln(cTest);           {print: k}
  cTest := ChUpper(cTest); {convert to upper case letter}
  Writeln(cTest);           {print: K}
end;
```

Str2HStr**Procedure**

```
procedure Str2HStr(var cs: String; var xBuf);
```

Parameter

csString.

Referenz to a `String`. String content to be copied into the array.

xBuf.

Referenz to an `array` of `Char`. Array into which the string is to be copied.

Return value

None.

Description

The procedure copies the content of the string `cs` byte by byte to the array `xBuf`. In case the number of bytes in the string is bigger than the size of the array `xBuf` the procedure stops copying when the array is full.

Remarks

Untyped parameter `xBuf`, allows to pass any type of `Char` array without having to do a type cast.

Example

```
procedure vMain;
var
  i32Count: Int32;
  sTest:    String;
  acTest:   array[0..2] of Char;
begin
  sTest := "ABCD";
  Str2HStr(sTest, acTest);
  for i32Count := 0 to 2 do
    Write(acTest[i32Count]); {print ABC}
  endfor
end;
```

5.2 Convert ByteString to String

Declaration

```
function Bin2Hex (bsBStr: ByteString): String;
```

Parameter

bsBStr.

ByteString. A binary string that is to be converted into a hexadecimal string representation.

Return value

String.

A hexadecimal string representation of the passed parameter.

Description

The function Bin2Hex converts binary values into hexadecimal strings.

Remarks

None

Example

```
procedure vMain;
var
  aTest: array[0..3] of Byte;
  csTest: String;
  bsTest: ByteString;

begin
  bsTest := 10;
  Writeln(bsTest);           {print: 10}
  csTest := Bin2Hex(bsTest); {convert into a hex string}
  Writeln(csTest);          {print: 0A}
end;
```

Bin2HexD**Declaration**

```
function Bin2HexD (bsBStr: ByteString; csDelim: String): String;
```

Parameter

bsBStr.

Byte string containing the values that are to be separated with a delimiter.

csDelim.

String. Character string that is used as delimiter string.

Return value

String.

String with a hexadecimal representation of the bytes delimited with the string provided with the csDelim parameter.

Description

The function Bin2HexD converts binary values into a hexadecimal string representation adding a delimiter between the single bytes. The delimiter characters can be set by the user.

Remarks

None.

Example

```
procedure vMain;
var
  csTest: String;
  bsTest: ByteString;
begin
  bsTest := BStrOf(255,5);           {creates ByteString containing 5 bytes
                                   with the value 255}
  csTest := Bin2HexD(bsTest,"-");   {convert the string and add delimiters}
  Writeln(csTest);                 {print: FF-FF-FF-FF-FF}
end;
```

5.3 General String and ByteString functions

Declaration

```
function Bottom (bcs: String; xoCount: tSize): String; bcstr;
```

Parameter

bcs.

ByteString or String. A string whose substring is to be returned.

xoCount.

Data type tSize equals Int32. Number of characters to return. Counting starts at the last character backwards.

Return value

String Or ByteString.

Extracted String or Byte-String.

Description

The function returns the last xoCount bytes of the string. All characters are returned in case xoCount is bigger than the size of the string.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  bsTest: ByteString;
  bsRes:  ByteString;
begin
  bsTest := BStrOf(255,5);      {creates a ByteString with 5 bytes
                               containing 255}
  bsTest[3] := 10;            {set the last but one value of the ByteString}
  bsRes := Bottom(bsTest,2);
  Writeln(bsRes);             {prints: (10,255)}
end;
```

Copy**Declaration**

```
function Copy (bcs: String; xolIndex, xoCount: tSize): String; bcstr;
```

Parameter

bcs.

String or ByteString. A string whose substring is copied and returned.

xolIndex.

Data type tSize equals Int32. xolIndex indicates the start position.

xoCount.

tSize. Number of bytes to copy.

Return value

String or ByteString.

The copied substring.

Description

The function copies a substring of xoCount bytes starting at index xolIndex.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  bsTest: ByteString;
  bsRes: ByteString;
begin
  bsTest := BStrOf(255,5);      {creates a ByteString with 5 bytes
                               containing 255}
  bsTest[3] := 10;            {set the last but one value of the ByteString}
  bsRes := Copy(bsTest,2,2);   {copy two bytes to bsRes starting at index 2}
  Writeln(bsRes);             {prints: (255,10)}
end;
```

Delete**Declaration**

```
procedure Delete (var bcs: String; xoIndex, xoCount: tSize); bcstr;
```

Parameter

bcs.

String or Byte-String as reference. A string which is to be shortened.

xoIndex.

Data type tSize equals Int32. xoIndex indicates the starting index for deletion.

Maximum xoCount - Abs(xoIndex) characters are deleted starting at position 0 if xoIndex < 0.

No substring is deleted if xoIndex >= Length(bcs) or xoCount <= 0.

The number of bytes to delete is decreased to Length(bcs) - xoIndex if needed.

xoCount.

tSize. Number of bytes to delete.

Return value

None.

Description

The procedure deletes a substring of xoCount bytes starting at position xoIndex.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  bsTest: ByteString;
begin
  bsTest := BStrOf(255,5); {creates a ByteString with 5 bytes containing 255}
  bsTest[3] := 10;        {set the last but one value of the ByteString}
  Delete(bsTest,0,2);     {deletes 2 bytes starting at index 0 in bsTest}
  Writeln(bsTest);       {prints: (255,10,255)}
end;
```

DelSpace

Declaration

```
procedure DelSpace(var cs: String; boSetBlank: Boolean);
```

Parameter

cs.

String as reference. A string whose whitespaces are to be deleted or replace with a blank.

boSetBlank.

Leading and trailing spaces are deleted if boSetBlank = false.

Additional whitespaces contained in the string are replaced with a blank if boSetBlank = true.

Return value

None.

Description

The procedure deletes spaces from the string. If boSetBlank is false only leading and trailing whitespaces are truncated. If boSetBlank is true additional whitespaces contained within the string are replaced with blanks.

Remarks

None.

Example

```
procedure vMain;
var
  csTest: String;

begin
  csTest := "  Test  string  "; {tab Test tab string tab}
  Writeln(csTest,"1");          {prints: "  Test  string  1"}
  DelSpace(csTest,false);      {deletes leading and trailing whitespaces}
  Writeln(csTest,"1");          {prints: "Test  string1"}
  DelSpace(csTest,true);       {replace all whitespaces with blanks}
  Writeln(csTest,"1");          {prints: "Test string1"}
end;
```

Insert**Declaration**

```
procedure Insert (bcsSource: String; bcs: String; xolIndex: tSize); bcstr;
```

Parameter

bcsSource.

String or ByteString as reference. String to be inserted.

bcs.

String or ByteString. String into which bcsSource is to be inserted.

xolIndex.

tSize equals Int32. Indicates the index where bcsSource is to be inserted.

Return value

None.

Description

The procedure inserts the string bcsSource into the string bcs at the position xolIndex. Nothing is inserted if xolIndex is bigger than the length of the string bcs. The string bcsSource is prepended to the string bcs if xolIndex is 0.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  sTest: String;

begin
  sTest := "ABCDEF";
  Insert("??", sTest, 1); {inserting ?? at position 1}
  Writeln(sTest);        {prints: A??BCDEF}
end;
```

Length

Declaration

```
function Length (bcs: String): ?t31Bit; bcsstr;
```

Parameter

bcs.

String or ByteString. A string, whose length in bytes is returned.

Return value

?t31Bit.

String length.

Description

The function returns the length of the string in bytes.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  sTest: String;

begin
  sTest := "ABC";
  Writeln(Length(sTest));    {prints: 3}
end;
```

Pos**Declaration**

```
function Pos (bcsSubstr: String; bcs: String; xolIndex: tSize): Int32; bcsStr;
```

Parameter

bcsSubstr.

String or ByteString. A string, whose position in bcs is returned.

bcs.

String or ByteString. A string, that is to be searched for the position of the substring bcsSubstr.

xolIndex.

tSize equals Int32. Start position for searching the substring bcsSubstr in bcs. The function searches backwards if a negativ index is passed starting at Length(bcs) - abs(xolIndex). The start position is decreased to Length(bcs) - Length(bcsSubstr) if necessary.

Return value

Int32.

Position of bcsSubstr in bcs. -1 is returned in case that the substring was not found, at least one of both strings is empty or xolIndex has an invalid value.

Description

The function returns the position of the substring bcsSubstr within the string bcs. Seaching starts at position xolIndex.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  sTest: String;

begin
  sTest := "Search the substring ";
  Writeln(Pos("substring",sTest,8)); {prints: 11}
```

Pos

```
Writeln(Pos("substring",sTest,12)); {prints: -1}
sTest := "abcdabcdeabcdef";
Writeln(Pos("abc",sTest,8));      {prints: 9}
Writeln(Pos("abc",sTest,-8));    {prints: 4}
end;
```

StrCompress**Declaration**

```
function StrCompress (csEscStr: String): String;
```

Parameter

csEscStr.

String. A String, that is to be encoded in ASCII.

Return value

String.

ASCII encoded string.

Description

The function encodes an string that contains ASCII code inclusive escape sequences using C conventions. (see example)

Remarks

You can look up the meaning of ASCII codes in the internet if necessary. Search for ASCII CODE using www.google.com

Example

(see also StrExpand (next page))

```
procedure vMain;
begin
  Writeln(StrCompress("\x41\102C"));    {prints: ABC}
  Writeln(StrCompress("\x32\102\nC")); {prints: 2B
                                         C}
end;
```

StrExpand

Declaration

```
function StrExpand (cs: String): String;
```

Parameter

cs.

String. A string that is to be converted into ASCII code.

Return value

String.

ASCII code of the string.

Description

Converts a string containing control characters into ASCII code using escape sequences in C convention.

Remarks

Control characters that have a C escape sequence notation ('\n' usw. except '\?') are converted according to C convention. The characters \x20 to \x7E (isprint()) and \x80 to \xFF (includes also !isASCII(), since real 7 bit systems are not relevant today) are passed through without processing. All other characters are converted into octal notation (always using 3 digits at the end or when an octal number follows).

The maximum length of the resulting string is 4*Length(cs).

Possibly less than three digits are used to represent characters in octal notation, thus you can not simply create strings containing octal characters by concatenating three digit strings. Anyway strings containing octal characters usually are not used for internal processing.

Three digits are always used at the end of a string, thus you can compose such strings without problems.

You can look up the meaning of ASCII codes in the internet if necessary. Search for ASCII CODE using www.google.com

Example

(see also StrCompress)

```
procedure vMain;
var
  csTest: String;
begin
```

StrExpand

```
csTest := "Tes\n";
Writeln(csTest," length: ", Length(csTest)); {prints: Tes
                                             t length: 5}
csTest := StrExpand (csTest);
Writeln(csTest," length :", Length(csTest)); {prints: Tes\n length: 6}
csTest := StrCompress (csTest);
Writeln(csTest," length :", Length(csTest)); {prints: Tes
                                             t length: 5}
end;
```

StrUpper**Declaration**

```
function StrUpper(cs:String): String;
```

Parameter

cs.

String. A string, whose characters are converted to upper case letters.

Return value

String.

String containing upper case letters.

Description

The function converts lower case letters into upper case letters. Other characters than lower case letters are not modified.

Remarks

None.

Example

```
procedure vMain;
var
  csRes: String;

begin
  csRes := StrUpper("TestSTring1$?"); {convert lower case to upper case}
  Writeln(csRes);                    {prints: TESTSTRING1$?}
end;
```

StrLower**Declaration**

```
function StrLower(cs:String): String;
```

Parameter

cs.

String. A string, whose characters are converted to lower case letters.

Return value

String.

String containing lower case letters.

Description

The function converts upper case letters into lower case letters. Other characters than upper case letters are not modified.

Remarks

None.

Example

```
procedure vMain;
var
  csRes: String;

begin
  csRes := StrLower("TestSTring1$?"); {convert upper case to lower case}
  Writeln(csRes);                    {prints: teststring1$?}
end;
```

SwapStrs

Declaration

```
procedure SwapStrs(var bcs1, bcs2: String); bcstr;
```

Parameter

bcs1.

String or ByteString as reference. String, that is copied to the second string.

bcs2.

String or ByteString as reference. String, that is copied to the first string.

Return value

None.

Description

The procedure swaps the content of two strings.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  sVar1: String;
  sVar2: String;

begin
  sVar1 := "String1";
  sVar2 := "String2";
  Writeln(sVar1, " ", sVar2); {prints : String1 String2}
  SwapStrs(sVar1, sVar2);    {swaps the content of the strings}
  Writeln(sVar1, " ", sVar2); {prints: String2 String1}
end;
```

Top**Declaration**

```
function Top (bcs: String; xoCount: tSize): String; bctr;
```

Parameter

bcs.

String or ByteString. A string whose substring is to be returned.

xoCount.

tSize equals Int32. Number of characters to be returned. Counting starts at the first character.

Return value

String.

String containing the first xoCount characters of bcs.

Description

The function returns the first xoCount bytes of the string.

Remarks

The function is marked with the key word BCSTR indicating that you can use the function with String and ByteString as parameter and return value.

Example

```
procedure vMain;
var
  csRes: String;

begin
  csRes := Top("String1",6); {get the first 6 characters}
  Writeln(csRes);           {prints: String}
end;
```

StrfStr**Functions**

```
function StrfStr (cs: String; i16Width: Int16; cMode: Char): String;
```

Parameter

cs.

String. A string, that is to be converted into another string.

i16Width.

Int16. Minimum width. The resulting string is longer than i16Width if i16Width is less than the size necessary to create the new string. The function uses left alignment when i16Width is negative and right alignment otherwise.

cMode.

Char. Indicates if and which delimiter to use (see example):

'\0' or '' (backslash + 0 or blank)

No delimiter.

'\ ', '"', '(' etc. (backslash + single quote, double quote, bracket, etc.)

Specified delimiter (see example).

Return value

String.

String with the minimum length of i16Width. The string starts and end with a delimiter if one was specified in the cMode parameter.

Description

Extends the length of a string. The output is left aligned (i16Width < 0) or right aligned (i16Width > 0). You can use cMode to specify a delimiter character. This delimiter is attached at the beginning and the end of the string.

Remarks

No special treatment is used for the delimiters within the string. You can convert '\ ' and '" with the function StrExpand.

StrfStr**Example**

```
procedure vMain;
begin

  Writeln(StrfStr("Test",10," "), "1"); {prints:      Test1}
  Writeln(StrfStr("Test",-10," "), "2"); {prints:Test      2}
  Writeln(StrfStr("Test",10,"@"), "3"); {prints:      @Test@3}
  Writeln(StrfStr("Test",5,"@"), "4");  {prints:@Test@4}
end;
```

vStrXlat**Declaration**

```
procedure vStrXlat(var cs:String; var ac: tacXlat; boRem0: Boolean);
```

Parameter

cs.

String as reference. A string, whose characters are to be converted according to a provided table.

ac.

tacXlat as reference. Array, that defines the conversion table. (tacXlat = array[Char] of Char;)

boRem0.

Boolean. '\0' and undefined characters are removed if boRem0 is true, otherwise they are undefined.

Return value

None.

Description

The procedure converts all characters of the string according to the passed array ((cs[i]:=ac[sc[i]]), with i representing a char variable). Characters in the string cs that have no corresponding entry in ac are undefined in the resulting string if boRem0 is false. '\0' and undefined characters are removed if boRem0 is true.

Remarks

None.

Example

```
procedure vMain;
var
  csTest:    String;
  acXlat:    tacXlat;

begin
  {simple encoding: replace letters by the following letter of the alphabet}
  acXlat["a"] := "b";
  acXlat["b"] := "c";
  acXlat["c"] := "d";
```

vStrXlat

```
{etc.....}
acXlat["z"] := "a";

csTest := "ab\0cz";
vStrXlat(csTest,acXlat,true); {encode the string and remove \0}
Writeln(csTest);             {prints: bcda}
end;
```

5.4 Functions to convert strings into values

Declaration

```
function Val (cs: String; xolIndex, xoCount: tSize; bRadix: Byte;
             var stResult: tstVal): ?t31Bit;
```

Parameter

cs.

String or ByteString. A string, whose content is to be converted into an numeric value.

xolIndex.

Integer variables or constants. Start position within the string for converting characters into numeric values (first character has the index 0).

xoCount.

tSize equals Int32. Number of characters to convert into a numeric value.

bRadix.

Base to use (see also the description of the structure stVal). The base 10 is used if an invalid radix is passed. (e.g. <= 1)

stVal.

tstVal as reference (see also the description of the structure stVal). Structure is filled with the extracted value and the used radix (base). Possible values are stVal.bRadix 0 (real) or 10 (int or enum)).

stVal.enBType (=nenBT7Bit..nenBTDWord,nenBTReal32,nenBTReal64) contains the data type of the value or nenBTNNone in case of an error.

The value is contained in stVal.i32, stVal.dw or stVal.r64 according to the parameters data type

Return value

?t31Bit.

The position of an error or the next position following to the processed characters.

Val**Description**

The function converts maximal xoCount characters into a numerical value starting at cs[xoIndex]. Length(cs)-xoIndex characters are converted starting at cs[xoIndex] if xoCount <= 0.

The result is the position of an error or the next position following to the processed characters. A Delimiter following the number is not considered as error, even when xoCount includes the delimiter. The base 10 is used if an invalid radix is passed. stResult.enBType contains the base type (or nenBTNone in case of an error). stResult.bRadix contains the used radix (or 0 with real) and the result is returned in stResult.i32, dw or r64. The supported data types match the data types of the compiler.

Remarks

(see also the description of the structure tstVal)

Example

```

procedure vMain;
var
  i32Pos:   Int32;
  i32LenCs: Int32;
  i32Count: Int32;
  i32PosAlt: Int32;
  bRadix:   Byte;
  xoCount:  tSize;
  csTest:   String;
  stVal:    tstVal;

begin
  i32Pos      := 0;           {start position in the string}
  i32PosAlt := 0;
  bRadix     := 10;        {base is 10}
  xoCount    := 3;         {convert 3 characters each time}
  csTest     := "25467844r"; {String to process}
  i32LenCs   := Length(csTest);
  {extract 3 valid characters each time. Abort if less or invalid characters}
  while i32Pos < i32LenCs do
    i32Pos := (Val(csTest, i32Pos, xoCount, bRadix, stVal));
    if (i32Pos <> i32PosAlt + 3) then
      Writeln("Error detected. Processing aborted.");
      break;
    endif;
    i32PosAlt := i32Pos;
    Writeln("Value: ", stVal.i32); {prints: Value: 254   }
                                   {           Value: 678   }
                                   {           Error detected ...}
  endwhile;
end;

```

StrCh2Int**Declaration**

```
function StrCh2Int (cs: String; xoPos: tSize): Int16;
```

Parameter

cs.

String. A string, containing a character whose ASCII code to returned.

xoPos.

tSize equals Int32. Position of the character that is to be convertet into ASCII.

Valid values for xoPos:

$0 \leq \text{xoPos} < \text{Length}(\text{cs}) = \text{StrCh2Int} := \text{Ord}(\text{cs}[\text{xoPos}])$.

Return value

Int16.

ASCII code of the character in case xoPos is valid.

Condition:

$0 \leq \text{xoPos} < \text{Length}(\text{cs}) = \text{StrCh2Int} := \text{Ord}(\text{cs}[\text{xoPos}])$

Otherwise -1 is returned.

Description

The function returns the ASCII code of the character at position xoPos of the string cs.

Remarks

You can also use `Ord(cs[xoPos])` in program code that ensures that $0 \leq \text{xoPos} < \text{Length}(\text{cs})$ is valid (e.g. for `xoPos := 0 to Length(cs)-1` do).

Example

```
procedure vMain;
var
  sTest:   String;

begin
  sTest := "ABCD";
  Writeln(StrCh2Int(sTest,0)); {prints: 65}
  Writeln(StrCh2Int(sTest,10)); {prints: -1}
end;
```

<x>Val**Declaration (<x>Val)**

```
function BVal (cs: String): Byte; external readby_by_st;  
function WVal (cs: String): Word; external readwd_wd_st;  
function DWVal (cs: String): DWord; external readlw_lw_st;  
function I8Val (cs: String): Int8; external readsh_sh_st;  
function I16Val(cs: String): Int16; external readin_in_st;  
function I32Val(cs: String): Int32; external readli_li_st;  
function R32Val(cs: String): Real32; external readsr_sr_st;  
function R64Val(cs: String): Real64; external readre_re_st;
```

Parameter

cs.

String. A string that is to be converted.

Return value

According to the function Byte, Word, DWord, Int8, Int16, Int32, Real32 and Real64.

Value that was converted from the String.

Description

The function converts a string to a numeric value if possible.

Remarks

Invalid characters e.g. like a D at the end of an Int32 string are ignored. 0 is returned if the invalid character is at the beginning or within the number. 0 is also returned when the range of the data type is exceeded.

Example

```
procedure vMain;  
var  
  sTest: String;  
  r64Val: Real64;  
begin  
  sTest := "32.45";  
  r64Val := R64Val(sTest); {converts a string into Real64}  
  r64Val := r64Val + 0.05; {32.45 + 0.05 = 32.5}  
  Writeln(r64Val);       {prints: 32.5}
```

BVal

```
end;
```

BVal**Declaration**

```
function BValh (cs: String): Byte; external readhby_by_st;  
function WValh (cs: String): Word; external readhwd_wd_st;  
function DWValh(cs: String): DWord; external readhlw_lw_st;
```

Parameter

cs.

String. A string containing a hexadecimal value that is to be converted into a numeric value.

Return value

According to the function Byte, Word, DWord. The numeric value that was extracted from the string.

Description

The function converts a hexadecimal string representation into a numeric value if possible.

Remarks

0 is returned if an invalid character is detected. In case that the range of the data type is exceeded the function returns the numeric value, build by starting on the right hand side and using up as much characters as possible to build the value without exceeding the range.

Example

```
procedure vMain;  
var  
  sTest: String;  
  wVal: Word;  
  
begin  
  sTest := "FFFF";  
  wVal := WValh(sTest);  
  Writeln(wVal);           {prints: 65535}  
  sTest := "F0001";       {exceeding the range}  
  wVal := WValh(sTest);  
  Writeln(wVal);           {prints: 1}  
  sTest := "GFF00";       {invalid character}  
  wVal := WValh(sTest);  
  Writeln(wVal);           {prints: 0}  
end;
```

BoVal**Declaration**

```
function BoVal (cs: String): Boolean; external readbo_bo_st;
```

Parameter

cs.

String. A string that is to be converted into a boolean value.

Return value

Boolean.

Boolean value extracted from the String.

Description

The function converts the string to a boolean value if possible.

Remarks

Leading and trailing whitespaces are removed and afterward case-sensitiv compared to "TRUE" resp. "FALSE".

false is returned in case that the string contains invalid characters.

Example

```
procedure vMain;
var
  sTest: String;
  boVal: Boolean;

begin
  sTest := " tRue "; {string to be converted to a Boolean value}
  boVal := BoVal(sTest); {convert string to Boolean value}
  Writeln(boVal); {prints: true}
  sTest := " tR ue "; {blank contained within the string true}
  boVal := BoVal(sTest); {convert string to boolean value}
  Writeln(boVal); {prints: false}
end;
```

5.5 Converting charaters to strings

Declaration

```
function StrOf (c: Char; xoCount: tSize): String;
```

Parameter

c.

Char. Content that is copied the signle bytes fo the string.

xoCount.

tSize equals Int32. Length of the string.

Return value

String.

String of the length xoCount bytes each containing content c.

Description

The function creates a string of the length xoCount bytes each containing content c.

Remarks

None.

Example

```
procedure vMain;
var
  csRes: String;

begin
  csRes := StrOf("A",5); {create a string with 5 bytes containing A}
  Writeln(csRes);       {prints: AAAAA}
end;
```

HStr2Str**Declaration**

```
function HStr2Str (var xBuf): String;
```

Parameter

xBuf.

array of char as reference. Characters to be copied into a string.

Return value

String.

String containing the characters from xBuf (until \0).

Description

The function copies the characters of the char array into a string (except \0).

Remarks

An untyped parameter is passed in order to be able to pass any type of char arrays without casting.

Example

```
procedure vMain;
var
  aTest:   array[0..4] of Char;
  cSign:   Char;
  i8Count: Int8;

begin
  for i8Count := 0 to 4 do
    aTest[i8Count] := "A";   {fill array of char}
  endfor
  Writeln(HStr2Str(aTest));  {prints: AAAAA}
end;
```

5.6 Converting values to strings

Declaration

```
function StrBo (bo: Boolean): String;
```

Parameter

bo.

Boolean. Value to be converted to a string.

Return value

String.

Representation of the value as string.

Description

The function converts a boolean value into a string representation.

Remarks

None.

Example

```
procedure vMain;
var
  csRes:  String;
  boTest: Boolean;

begin
  boTest := true;
  csRes  := StrBo(boTest); {converts a boolean value to a string}
  Writeln(csRes);         {prints: true}
end;
```

StrbB**Declaration**

```
function StrbB (b: Byte): String;  
function StrbW (w: Word): String;  
function StrbDW(dw: DWord): String;
```

Parameter

According to the function b, w or dw.

According to the function Byte, Word or DWord.

Return value

String.

Representation of the value as binary string.

Description

The function converts a value into a string in a binary representation. The resulting string always has the length of the data type in binary representation. StrbW returns a string with a length of 16 bytes.

Remarks

None.

Example

```
procedure vMain;  
var  
  csRes: String;  
  wTest: Word;  
  
begin  
  wTest := 3;  
  csRes := StrbW(wTest); {convert a variable of data type word to a string}  
  Writeln(csRes);       {prints: 0000000000000011}  
end;
```

StrhB**Declaration**

```
function StrhB (b: Byte): String; external strhby_st_by;
function StrhW (w: Word): String; external strhwd_st_wd;
function StrhDW(dw: DWord): String; external strhlw_st_lw;
```

Parameter

According to the function b, w or dw.

According to the function Byte, Word or DWord.

Return value

String.

Hexadecimal representation of the value as string

Description

The function converts a value into string in a hexadecimal representation. The resulting string has the length that is used to represent the value in hexadecimal notation. StrhW returns a string with a length of 4 bytes.

Remarks

None.

Example

```
procedure vMain;
var
  csRes: String;
  wTest: Word;

begin
  wTest := 255;
  csRes := StrhW(wTest); {convert a variable of data type word into a string
                        using hexadecimal representation of the value}
  Writeln(csRes);       {prints: 00FF}
end;
```

StrDW**Declaration**

```
function StrDW (dw: DWord): String; external strlw_st_lw;  
function StrI32(i32: Int32): String; external strli_st_li;  
function StrRe (r64: Real64): String; external strre_st_re;  
function StrPtr(pv: Pointer): String; external strpt_st_pt;
```

Parameter

According to the function dw, i32, r64 or pv.

According to the function Byte, Word, DWord or Pointer.

Return value

String.

Representation of the value as string.

Description

The functions convert a value into a string. The length of the resulting string equals the number of characters used to represent the value (see example). Pointers always use 8 bytes with an @ and a colon prepended.

Remarks

None.

Example

```
procedure vMain;  
var  
  csRes: String;  
  dwTest: DWord;  
  
begin  
  dwTest := 32767;  
  csRes := StrDW(dwTest); {convert a value of type DWord into a String}  
  Writeln(csRes); {prints: 32767}  
  Writeln(Length(csRes)); {prints: 5}  
  csRes := StrPtr(@dwTest); {convert a pointer into a String}  
  Writeln(csRes); {prints: @:00569923}  
end;
```

StrVal**Declaration**

```
function StrVal(var stVal: tstVal): String;
```

Parameter

stVal.

tstVal as reference (see also the description of the structure tstVal). Value of the structure tstVal, that is to be converted into a string.

Return value

String.

Representation of the value as string.

Description

The function converts a value in a tstVal structure into a string. The function accounts the radix of stVal when formatting the string (using base 2, 10 or 16. The number of digits is determined using the base type.

Remarks

An empty string is returned in case that the value in stVal is nenBTNone.

Example

(see also structure tstVal and the description of untyped parameters in chapter 16.3)

```
csValueAsString := StrVal(stVal); {Initialize stVal}
```

StrfDW**Declaration**

```
function StrfDW (dw: DWord; i16Width: Int16; cMode: Char): String;
```

```
function StrfI32(i32: Int32; i16Width: Int16; cMode: Char): String;
```

Parameter

dw or i32.

DWord or Int32. A value that is to be converted into a string.

i16Width.

Int16. Minimum width. The returned string is larger than the specified minimum width when more characters are required to represent the value. The function uses left alignment when $i16Width < 0$.

cMode.

Char. Desired representation of the string.

modes:

" " : Prepend a space to positive values (space used for algebraic sign).

"+" : Always add the algebraic sign ('+').

", " : Use group delimiter (billion, million, thousand)

"0" : Fill with '0' (ignored when Width < 0).

"\0" : Normal representation of the string.

Return value

String.

Representation of the value as formatted string.

Description

The function converts a value into a string with the minimum length i16Width. The function uses left alignment when i16Width is negative and right alignment otherwise. You can specify different representations with the cMode parameter.

Remarks

None.

StrfDW**Example**

```
procedure vMain;
var
  dwTest: DWord;
begin
  dwTest := 65535;
  Writeln(StrfDW(dwTest, -10, ", "));
end;
```

StrfbDW**Declaration**

```
function StrfbDW (dw: DWord; i16Width, i16Digits: Int16; cMode: Char): String;
```

```
function StrfbI32(i32: Int32; i16Width, i16Digits: Int16; cMode: Char): String;
```

Parameter

dw or i32.

DWord or Int32. A value to be converted into a binary string representation.

i16Width.

Int16. Minimum width. The returned string is larger than the specified minimum width when more characters are required to represent the value. The function uses left alignment when $i16Width < 0$ and right alignment otherwise.

i16Digits.

Int16. Specifies that the minimum digits used to represent the value is a multiple of four ($-4 = 4, 8, 12, \dots$) or a multiple of eight ($-8 = 8, 16, 24, \dots$). To achieve this zeros are padded to the left side of the binary number. In case an other value than -4 or -8 is passed, only the necessary number of digits is used.

cMode.

Char. Desired representation of the string.

Modes:

"b" : Appends the character b to the binary string.

"\0" : Normal representation of the binary string.

Return value

String.

Representation of the value as binary string.

Description

The function converts a value into a binary string representation with the minimum length i16Width. The function uses left alignment when i16Widths is negativ and right alignment otherwise. i32Digits specifies that the minimum digits used to represent the value is a multiple of four ($-4 = 4, 8, 12, \dots$) or a multiple of eight ($-8 = 8, 16, 24, \dots$). To achieve this zeros are padded to the left side of the binary number. In case an other

StrfbDW

value than -4 or -8 is passed, only the necessary number of digits is used. You can specify different representations with the cMode parameter.

Remarks

None.

Example

```
procedure vMain;
var
  dwTest: DWord;

begin
  dwTest := 1023;
  Writeln(StrfbDW(dwTest,20,1,"b"));   {prints:      111111111b}
  Writeln(StrfbDW(dwTest,20,-4,"b")); {prints:      0011111111b}
  Writeln(StrfbDW(dwTest,20,-8,"b")); {prints: 000000111111111b}
  Writeln(StrfbDW(dwTest,20,-8,"\0")); {prints: 000000111111111}

end;
```

StrfhDW**Declaration**

```
function StrfhDW (dw: DWord; i16Width, i16Digits: Int16; cMode: Char): String;
```

Parameter

dw.

DWord. A value that is to be converted into a hexadecimal string.

i16Width.

Int16. Minimum width. The returned string is larger than the specified minimum width when more characters are required to represent the value. The function uses left alignment when $i16Width < 0$ and right alignment otherwise.

i16Digits.

Int16. Specifies that the minimum digits used to represent the value is a multiple of two ($-2 = 2, 4, 8$) or a multiple of four ($-4 = 4, 8$). To achieve this zeros are padded to the left hand side of the hexadecimal number. In case an other value than -2 or -4 is passed, only the necessary number of digits is used.

cMode.

Char. Desired representation of the string.

Modes:

"X", "x" : 0x + number ('x' uses 'a'-'f' and 'X' uses 'A'-'F').

"H", "h": number + h (a 0 is prepended when necessary.)

"\$" : \$ + number.

"\0" : No prefix or suffix is used.

Return value

String.

Representation of the value as hexadecimal string.

Description

The function converts a value into a hexadecimal string representation with the minimum length *i16Width*. The function uses left alignment when *i16Width* is negative and right alignment otherwise. *i16Digits* specifies that the minimum digits used to represent the value is a multiple of two ($-2 = 2, 4, 8$) or a multiple of four ($-4 = 4, 8$). To achieve this zeros are padded to the left hand side of the hexadecimal number. In case

StrfhDW

an other value than -2 or -4 is passed, only the necessary number of digits is used. You can specify different representations with the cMode parameter.

Remarks

None.

Example

```
procedure vMain;
var
  dwTest: DWord;

begin
  dwTest := 65536;
  Writeln(StrfhDW(dwTest,15,1,"X")); {prints:      0x10000}
  Writeln(StrfhDW(dwTest,15,-2,"X")); {prints:      0x010000}
  Writeln(StrfhDW(dwTest,15,-4,"X")); {prints: 0x00010000}
end;
```

StrfRe**Declaration**

```
function StrfRe (r64: Real64; i16Width, i16Prec: Int16; cMode: Char): String;
```

Parameter

r64.

Real64. A numeric value that is to be converted to a string representation.

i16Width.

Int16. Minimum width. The returned string is larger than the specified minimum width when more characters are required to represent the value. The function uses left alignment when i16Width < 0 and right alignment otherwise.

i16Prec.

Number of decimal places. Limited to the maximum of 39 decemal places.

cMode.

Char. Desired representation of the string.

Modes:

"f": Use floating point notation (if possible).

"+": Like "f", but always include the sign (+ or -).

",";": Like "f", but uses group delimiter (billion,million,thousand)

"e","E": Using exponential format.

"g","G": Automatic format selection (f- resp. e-).

Windows uses f-format with exponent -4.. prec-

" ": Like "f", but prepend a space to positiv values (' ' or '-').

"\0": Like "g".

Return value

String.

Representation of the numeric value as string.

StrfRe**Description**

The function converts a numeric value into a string representation with the minimum length `i16Width`. The function uses left alignment when `i16Width` is negative and right alignment otherwise.

`i16Prec` specifies the number of decimal places.

You can specify different representations with the `cMode` parameter.

Remarks

It is not reasonable to print too many digits (and DII MSVC6 crashes when you try to use `printf("%0.320f", 1.7e+308)`). This is the reason why `i16Prec` is limited from 0 to 39. Furthermore the format changes from floating point notation to exponential notation automatically. (Remark: see formatted output in every good C programming book.)

Example

```
procedure vMain;
var
  r64Test: Real64;

begin
  r64Test := 12.5894;
  Writeln(StrfRe(r64Test, 10, 5, "+")); {prints: +12.58940}
  Writeln(StrfRe(r64Test, 10, 2, "+")); {prints:   +12.59}
end;
```

StrfPtr**Declaration**

```
Function StrfPtr(pv: Pointer; i16Width: Int16): String; external strpt_st_pt_in;
```

Parameter

pv.

Pointer. A pointer that is to be converted to a string.

i16Width.

Int16. Minimum width. The returned string is larger than the specified minimum width when more characters are required to represent the value. The function uses left alignment when $i16Width < 0$ and right alignment otherwise.

Return value

String.

Representation of the address as string.

Description

The function converts a pointer into a string representation with the minimum length i16Width. The function uses left alignment when i16Widths is negativ and right alignment otherwise.

Example

```
procedure vMain;
var
  pi32Test: ^Int32;

begin
  New(pi32Test);
  pi32Test^ := 100;
  Writeln(StrfPtr(pi32Test,15)); {prints:      @:00974014}
  Dispose(pi32Test);
end;
```

Strf**Declaration**

```
function Strf (..): String;
```

Parameter

..

Variable parameter list. You have to specify following parameters for each variable:

Variable: Unstructured variable or constant.

Width: Integer value. Number of characters used for the variable.

Precision: Integer value. precision, i.e. the number of decimal places (see example).

Mode: `String`. Representation mode. Always has to be specified at the end of the list. You can also omit parameters `Strf(l,,j,'X')`. Mode "M" truncates preceding module names and type identifiers (nen or en).

Mode "H" forces hexadecimal string representation of the value.

Mode "B" forces binary string representation of the value.

Return value

`String`. Formated string, consisting of the passed variables.

Description

The function converts unstructured variables and constants into a formatted string. Additional to the variable you can specify the desired width, the precision and the representation mode.

Remarks

<?> is inserted into the returned string when an invalid parameter or data type is passed to the function and the error code is set to EINVALID.

In the non decimal modes, variables have either the prefix ("0x","\$") or the suffix ("h","B").

Example

```
procedure vMain;
var
  csRes: String;
begin
  csRes := Strf(true,1,1,,32.2563,8,4,, "ABC",4,3); {convert variable to
                                                    string}
  Writeln(csRes); {prints: true   32.26
                  ABC}
  csRes := Strf(0xFF,3,0,"H",0x01,4,2,"B"); {convert variable to
```

Strf

```
Writeln(csRes);  
end;
```

```
string}  
{prints: 0FFh 01B}
```

Move2Str**Declaration**

```
procedure Move2Str (var xBuf; var bcs: String; xoCount: tSize); bcstr;
```

Parameter

XBuf.

As reference. Any data type except `String` und `ByteString`.

bcs.

`String` or `ByteString`. String that is copied to.

xoCount.

`tSize` equals `Int32`. Number of bytes, that are to be copied from xBuf.

`SizeOf(xBuf)` are copied when `xoCount < 0`.

Return value

None.

Description

The procedure copies `xoCount` bytes from `xBuf` in `bcs`. `SizeOf(xBuf)` are copied when `xoCount < 0`.

Remarks

The size of the object is retrieved automatically, based on the real object type when `xoCount < 0` instead of using the current (static) type.

The data types `String` and `ByteString` are not allowed as data source.

When the source is a `HeapBlock`, `xoCount` is limited to the size of the `HeapBlock`.

The procedure is marked with the key word `BCSTR`. This allows to use the data types `String` and `ByteString` as parameter and return value.

Example

```
procedure vMain;
var
  abTest: array[0..2] of Byte;
  bsRes: ByteString;
```

Move2Str

```
begin
  abTest[0] := 11;
  abTest[1] := 22;
  abTest[2] := 33;

  Move2Str(abTest,bsRes,-1); {copy sTest to sRes }
  Writeln(bsRes);           {prints: (11,22,33)}

  Move2Str(abTest,bsRes,2); {copy 2 bytes from sTest to sRes }
  Writeln(bsRes);           {prints: (11,22)}
end;
```

5.7 Special ByteString functions

Declaration

```
function BStrOf (b:Byte; xoCount:tSize): ByteString;
```

Parameter

b.

Byte. Content of the byte string.

XoCount.

Data type `tSize` equals `Int32`. The length of the byte string to be created.

Return value

ByteString.

Byte string of the length `xoCount` and each byte containing `b`.

Description

The function creates a byte string with the length `xoCount` and each byte containing `b`.

Remarks

None

Example

```
procedure vMain;
var
  bsTest: ByteString;
begin
  bsTest := BStrOf(255,5); {creates a ByteString with 5 bytes containing 255}
  Writeln(bsTest);       {prints: (255,255,255,255,255)}
end;
```

AB2BStr**Declaration**

```
function AB2BStr (var xBuf): ByteString;
```

Parameter

xbuf.

As reference. Either an element of a byte array or a byte array.

Return value

ByteString.

Byte string containing the values of the array.

Description

The function copies a character of a byte array or the whole byte array to a byte string.

Remarks

None

Example

```
procedure vMain;
var
  aTest: array[0..3] of Byte;
  bsTest: ByteString;

begin
  aTest[2] := 10;
  bsTest := AB2BStr(aTest[2]); {copy the 3. character to the ByteString}
  Writeln(bsTest);           {prints: (10)}
  bsTest := AB2BStr(aTest);  {copy the array to the ByteString}
  Writeln(bsTest);           {prints: (0,0,10,0)}
end;
```

Hex2Bin**Declaration**

```
function Hex2Bin (csHexStr: String): ByteString;
```

Parameter

csHexStr.

String. Hexadecimal string, that is to be converted into a byte string.

Return value

ByteString.

A byte string with the converted bytes from the hexadecimal string.

Description

The function converts a hexadecimal string into a byte string. An empty string is returned when the parameter is invalid (e.g. E).

Remarks

The last character is ignored when an odd number of characters is passed. (see the example).

Example

```
procedure vMain;
var
  csHex: String;

begin
  csHex := "A0FF";           {hexadecimal string A0FF}
  Writeln(Hex2Bin(csHex));  {print result: (160,255)}
  csHex := "HFF";          {invalid string}
  Writeln(Hex2Bin(csHex));  {print result: ()}
  csHex := "A3F";          {hexadecimal string AFF}
  Writeln(Hex2Bin(csHex));  {print result: (163) - the last F is ignored}
end;
```

5.8 Struktur QWord

Structure

```
QWord = record      {common unsigned 64 bit data type}
  ?dwLow:   DWord; {Remarks: Since the compiler will support this data type
                  in future, the type prefix ("tst") is omitted}
  ?dwHigh:  DWord; {and the elements are not declared public!}
end;
tpQWord = ^QWord;
```

Description

The structure `QWord` is a 64-Bit data type consisting of a `Low-DWord` and a `High-DWord`. The elements of the structure can not be accessed directly. This is intentional since the compiler will support this data type in future. You can work with the `QWord` data type using the provided functions which are explained in the following section.

5.9 QWord functions

Input/Output functions frequently use the data type `QWord`. Therefore we present some functions to manipulate `QWord` data types.

Declaration

```
procedure SetQW(var qw: QWord; dwHigh, dwLow: DWord);
```

Parameter

`qw`.

Referenz to a `QWord`.

`dwHigh`.

`DWord`. Value containing the upper four bytes to be written to `qw`.

`dwLow`.

`DWord`. Value containing the lower four bytes to be written to `qw`.

Return value

None.

Description

The procedure composes a `QWord` out of the `DWords` `dwHigh` and `dwLow`.

Remarks

None.

Example

```
procedure vMain;
var
  dwHigh: DWord;
  dwLow:  DWord;
  qwRes:  QWord;

begin
  dwHigh := 1;
  dwLow  := 65535;
```

SetQW

```
SetQW(qwRes,dwHigh,dwLow); {assign the QWord}  
Writeln(qwRes);           {print: (?dwLow:65535;?dwHigh:1}  
end;
```

SetHDWofQW**Declaration**

```
procedure SetHDWofQW(var qw: QWord; dwHigh: DWord);
```

Parameter

qw.

Referenz to a QWord.

dwHigh.

DWord. Value containing the upper four bytes to be written to qw.

Return value

None.

Description

The procedure replaces the upper four bytes of the QWord qw with the value of the DWord dwHigh.

Remarks

None.

Example

```
procedure vMain;
var
  dwHigh: DWord;
  dwLow: DWord;
  qwRes: QWord;

begin
  dwHigh := 1;
  dwLow := 65535;
  SetQW(qwRes, dwHigh, dwLow); {initialize the QWord}
  Writeln(qwRes);             {print: (?dwLow:65535;?dwHigh:1}

  SetHDWofQW(qwRes, 10);      {dwHigh = 1 is replaced by dwHigh = 10}
  Writeln(qwRes);             {print: (?dwLow:65535;?dwHigh:10}

end;
```

SetLDWofQW**Declaration**

```
procedure SetLDWofQW(var qw: QWord; dwLow: DWord);
```

Parameter

qw.

Referenz to a QWord.

dwLow.

DWord. Value containing the lower four bytes to be written to qw.

Return value

None.

Description

The procedure replaces the lower four bytes of the QWord qw with the value of the DWord dwLow.

Remarks

None.

Example

```
procedure vMain;
var
  dwHigh: DWord;
  dwLow: DWord;
  qwRes: QWord;

begin
  dwHigh := 1;
  dwLow := 65535;
  SetQW(qwRes, dwHigh, dwLow); {initalize the QWord}
  Writeln(qwRes);             {print: (?dwLow:65535;?dwHigh:1}

  SetLDWofQW(qwRes, 32767);   {dwLow = 65535 is replaced by dwLow = 32767}
  Writeln(qwRes);             {print: (?dwLow:32767;?dwHigh:1}

end;
```

GetHDWofQW**Declaration**

```
function GetHDWofQW(qw: QWord): DWord;
```

Parameter

qw.

QWord.

Return value

DWord.

The upper DWord (High-DWord) of the passed QWord qw.

Description

The function returns the upper DWord of the QWord qw. The data type QWord has the size of 8 bytes.

Remarks

None.

Example

```
procedure vMain;
var
  dwHigh: DWord;
  dwLow: DWord;
  qwRes: QWord;

begin
  dwHigh := 1;
  dwLow := 65535;
  SetQW(qwRes, dwHigh, dwLow); {initialise the QWord}
  Writeln(GetHDWofQW(qwRes)); {print: 1}
end;
```

GetLDWofQW**Declaration**

```
function GetLDWofQW(qw: QWord): DWord;
```

Parameter

qw.

QWord.

Return value

DWord.

The lower DWord (Low-DWord) of the passed QWord qw.

Description

The function returns the lower DWord of the passed QWord qw. The data type QWord has the size of 8 bytes.

Remarks

None.

Example

```
procedure vMain;
var
  dwHigh: DWord;
  dwLow: DWord;
  qwRes: QWord;

begin
  dwHigh := 1;
  dwLow := 65535;
  SetQW(qwRes, dwHigh, dwLow); {initialise the QWord}
  Writeln(GetLDWofQW(qwRes)); {print: 65535}
end;
```

6 File I/O functions

Taken from the pool.pli library.

6.1 Introduction

This section describes the functions and procedures that are used to work with files. For beginners who did not previously program any file access, we will briefly explain the necessary basics in the following:

At this point we will make a distinction between two types of files.

There are the binary files, in which the data is stored as hexadecimal byte values. To view this data you generally use a Hex editor. Bitmaps and the registry file in Windows are typical examples of binary files.

And then there are text files, in which characters are stored; these characters can be viewed using a text editor such as Notepad. The initialization files in Windows (.ini.) and "regular" text files with the ending .txt are typical examples.

In order to be able to work with a file, it has to be opened. Files are opened with the functions FOpen (for binary files) or TOpen (for text files).

The name (and possibly the path) of the file that is to be opened, an empty file handle, and the access mode are transferred to these functions. If a file with the specified name does not exist, it is automatically created.

Important:

Possibly required headers (e.g., with bitmaps) are not written automatically. (A file header is an area (normally at the beginning of the file), in which the information on the file (size, compression, file type etc.) is stored).

A file handle is a structure that includes a pointer to a file. It is transferred as a reference and used in additional function calls, in order to identify the opened file. Since the transfer of an invalid handle to a function can lead to serious program errors, it is necessary to check that the file was successfully opened by testing the return value. A structure is used as a file handle in POOL that contains, in addition to the file name, the selected access type and the position of the file pointer.

A distinction is made between the access modes read, write, or read and write. A distinction in writing is also made between overwriting and appending.

In the first case already existing data are overwritten starting at the beginning of the file. In the second case new data is appended behind the already existing file. If a file is open for writing and reading, the functions FFlush or TFlush should be called before reading from the file. They ensure that the data is written into the file from the internal buffer and that it is really the current data that is accessed when reading.

After a file was opened successfully, the content can be accessed depending on the mode. The `pool.pli` library supports the user with a number of functions that can be used to read and write data.

The access to binary files is usually done byte by byte, while either the individual bytes or any number of bytes are read or written. To buffer this data we recommend using the data type `ByteString` (see also section 5.7).

The access in text files is done based on individual characters or lines. Besides, there are functions that can be used to read or write certain data types (e.g., `Int32`, `Char`, `Real64`, etc.).

How does the function know, where data has to be read or written in the file? The solution is the so-called file pointer. A file pointer is a pointer to the current position within the file. The file pointer is located in the structure of the file handle and is always passed as a reference when file functions are called.

The file pointer points to the beginning of the data after opening the file (except when opening files with the access mode `append`; here the pointer points behind the existing data). If one or several bytes (or characters) are read or written, the file pointer is incremented and then points to the position behind this data. As a result, the entire file can be processed step by step. The end of the file can be detected using the `FEOF` (binary files) and `TEOF` (text files) functions. The examples on the individual functions will show what this can look like in practice.

There is a separate function that can be used to calculate the size of a file. Calculating the size is helpful whenever the entire file is to be read at once and intermediately stored in a buffer (usually of data type `ByteString`).

The position of the file pointer can be requested and manipulated via functions. Both an absolute and a relative position can be requested.

Reading a bitmap file is a typical application of the positioning of the file pointer. The position of the actual image data is calculated using the specifications in the header (first data in the file) and the file pointer is positioned to this data. Next, the image data can be buffered in a `Byte` string and processed via a loop.

The file should be closed as soon as you no longer need it. To do this, pass the file handle to the function that is used to close. You can check the success of this operation using the return value of the function. The file is automatically saved during closing. Data that is still in the internal writing buffer is written into the file beforehand.

For further details on individual functions please see the following descriptions.

General Note

All file names in POOL are separated using `/` instead of `\`. In general, it is not permitted to use absolute path specifications. The use of the environmental variable, which contains absolute paths, is an exception to this rule.

FOpen

6.2 Binary files

Declaration

```
function FOpen (fi: tFile; csName: String; enFMode: tenFMode): Boolean;  
  ifr;
```

Parameter

fi.

Referenz to tFile. File handle.

csName.

String. Name of the file to be opened.

enFMode.

tenFMode. The desired access modes.

Allowed access modes:

nenFMNone: Open without any access rights.

nenFMRead: Open file for reading.

nenFMUpdate: Open file for reading and writing. File pointer points to the beginning.

nenFMWrite: Open file for writing. File pointer points to the beginning of the file.

nenFMAppend: Open file for writing. File pointer points to the end of the file.

Return value

Boolean.

True, when the file was opened successfully.

False, in case the file name is missing, Fmode is invalid or fopen fails.

The syntax allows to ignore the return value.

Description

The function opens a file in binary mode. The access mode is specified using the parameter enFMode. The modifier ifr allows to use the the function as procedure i.e. ignore the return value. I case you try to open a non existent file for reading the FOpen function returns false and an invalid file handle.

FOpen**Remarks**

The modifier `if r` indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  if !FOpen(hFile, csFileNamePar, nenFMRead) then {open the file}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
  Writeln(hFile);                       {print content of hFile
                                         structure}
  if !FClose(hFile) then                {close file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
end;
```

FClose**Declaration**

```
function FClose (var fi: tFile): Boolean; ifr;
```

Parameter

fi.

Referenz to `tFile`. File handle to the file to be close. (see also `FOpen`).

Return value

Boolean.

True in case the file was closed successfully.

False in case of an invalid file handle or `fclose` failed.

The syntax allows to ignore the return value.

Description

The function closes a file, previously saving it.

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:         tFile;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  if !FOpen(hFile, csFileNamePar, nenFMRead) then {open the file}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
  Writeln(hFile);                       {print content of hFile}

  if !FClose(hFile) then                 {close the file}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
end;
```

BlockRead

Declaration

```
function BlockRead (var fi: tFile; var xBuf; xoCount: tSize): ?t31Bit;
```

Parameter

fi.

Referenz to tFile. File handle to an open file (see also FOpen).

xBuf.

All simple data types. Memory buffer for the data to be read.

xoCount.

Number of bytes to be read. In case xoCount < 0, SizeOf(xBuf) bytes are read from the file.

In case xBuf is a heap block, xoCount is limited to the size of the heap block.

Return value

?t31Bit.

Number of bytes read without error and zero in case of an invalid file handle.

Description

The function copies data from the file into the buffer xBuf. The size of xBuf is arbitrary. Often it is useful to use a buffer of data type ByteString. The number of bytes to read is determined with the parameter xoCount. In case xoCount is negative, the number of bytes to read is determined by the size of xBuf parameter, which is the filled with data from the file.

The start position of the bytes to read is indicated by the file pointer. After the bytes are read the file pointer is set behind the last read byte.

Remarks

None

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
```

BlockRead

```
    i32Res:          Int32;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  if !FOpen(hFile,csFileNamePar,nenFMRead) then {open the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  i32Res := BlockRead(hFile,bTest,-1);   {read the first byte}
  Writeln(bTest);                        {print the first byte}
  i32Res := BlockRead(hFile,bTest,-1);   {read the second byte}
  Writeln(bTest);                        {print the second byte}
  if !FClose(hFile) then                 {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
end;
```

BlockWrite

Declaration

```
function BlockWrite (var fi: tFile; var xBuf; xoCount: tSize): ?t31Bit;
```

Parameter

fi.

Referenz to tFile. File handle to a file opened with write access (see also FOpen).

xBuf.

All simple data types. Data to be stored into the file.

xoCount.

Number of bytes to write. In case xoCount < 0, SizeOf(xBuf) bytes are written from the file.

In case xBuf is a heap block, xoCount is limited to the size of the heap block.

Return value

?t31Bit.

Number of bytes written to the file without error. Zero is returned in case of an invalid file handle.

Description

The function writes the data from the buffer (xBuf) into the file. The size of the data to write is specified with the parameter xoCount. In case xoCount is negative, the number of bytes to write is determined by the size of xBuf parameter and then the data in the buffer xBuf is written to the file.

The start position of the bytes to be written is indicated by the file pointer. After the bytes are written the file pointer is set behind the last byte written.

Remarks

None

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
```

BlockWrite

```
    i32Res:          Int32;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  bTest := 255;
  if !FOpen(hFile,csFileNamePar,nenFMWrite) then {open the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  i32Res := BlockWrite(hFile,bTest,-1);   {write a byte}
  if !FClose(hFile) then                  {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif
end;
```

FEOF**Declaration**

```
function FEOF (var fi: tFile): Boolean;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

Return value

Boolean.

false, in case the file pointer does not point to the end of the file.

true, in case the file pointer points to the end of the file.

Description

The function returns false, until the end of the file is reached.

Remarks

The return value is also false in case an invalid file handle is passed. Using a loop to detect the end of the file can result in an endless loop.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
  i32Res:        Int32;
  qwPos:        QWord;
begin
  csFileNamePar := "Test.bmp";           {set the file name}
  if !FOpen(hFile, csFileNamePar, nenFMRead) then {open the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  while (FEOF(hFile) = false) do        {loop until the end of the
                                        file is not reached}
    if !(FGetPos (hFile, qwPos)) then   {get position}
      Writeln(GetErrorMsg(GetError));
      return;
    endif;
    Writeln(qwPos);                     {print position}
    i32Res := BlockRead(hFile, bTest, -1); {read next byte}
  endwhile
  FClose(hFile);                        {close the file - error
```

FEOF

```
end;
```

```
Handling omitted}
```

FFlush

Declaration

```
function FFlush (var fi: tFile): Boolean; ifr;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

Return value

Boolean.

false when passing an invalid file handle or fflush failed.

true, when the internal buffer is written to the file successfully.

Description

The function writes all internal buffered data to the file. Use this function when you opened a file with read/write access. This function assures that modified data is written to the file, so you can access the actual data.

Remarks

The modifier ifr indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
  i32Res:       Int32;
  qwPos:        QWord;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  bTest := 5;
  SetQW(qwPos, 0,0);                    {set position}
  if !FOpen(hFile,csFileNamePar,nenFMUpdate) then {open file with read/write
                                           access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  i32Res := BlockWrite(hFile,bTest,-1);  {write to the file}
  if !(FFlush(hFile)) then              {write file buffer to disk}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
```

FFlush

```
if !(FSetPos(hFile,qwPos)) then           {set file pointer}
    Writeln(GetErrorMsg(GetError));      {print error message}
endif;
i32Res := BlockRead(hFile,bTest,1);      {read data form file}
Writeln(bTest);
if !FClose(hFile) then                   {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
endif
end;
```

FSeek**Declaration**

```
function FSeek (var fi: tFile; xoOffs: tFOffs; enFOrg: tenFOrg): Boolean;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

xoOffs.

tFOffs equals Int32. File pointer offset.

enFOrg.

tenFOrg Indicates the origin used to set the file pointer.

nenSeekSet Set the file pointer relativ to the beginning of the file.

nenSeekCur Set the file pointer relativ to the current position.

nenSeekEnd Set the file pointer relativ to the end of the file.

Return value

Boolean.

false in case of an invalid file handle or when FSeek failed.

true if file pointer was positioned successfully.

Description

The function repositions the file pointer xoOffs bytes from the origin. Possible values for the origin are the beginning (enFOrg = nenSeekSet), the current position (enFOrg = nenSeekCur) or the end of the file (enFOrg = nenSeekEnd).

Remarks

None.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:         tFile;
  bTest:         Byte;
  i32Res:        Int32;
```

FSeek

```
begin
  csFileNamePar := "Test.bmp";           {set the file name}
  if !FOpen(hFile,csFileNamePar,nenFMRead) then {open file with read access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  if !(FSeek (hFile,10,nenSeekCur)) then   {set the file pointer 10
                                           bytes ahead to the current
                                           position}
    Writeln(GetErrorMsg(GetError));       {print error message}
    return;
  endif;
  i32Res := BlockRead(hFile,bTest,-1);    {read data at the new
                                           position}
  Writeln(bTest);                         {print the data}
  FClose(hFile);                          {close the file - error
                                           Handling omitted}
end;
```

FSize**Declaration**

```
function FSize (var fi: tFile; var qwSize: QWord): Boolean;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

qwSize.

QWord. Retrieves the size of the file in bytes.

Return value

Boolean.

false in case of an invalid file handle or when FSize failed.

true if successful.

Description

The function retrieves the size of the file in bytes. The file size is returned in the variable qwSize which is passed to the function as referenz.

Remarks

None

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
  i32Res:       Int32;
  qwSize:       QWord;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  SetQW(qwSize, 0,0);

  if FOpen(hFile,csFileNamePar,flenFMRRead) then {open file with read access -
    error handling omitted}
    if FSize(hFile,qwSize) then           {retrieve file size in bytes}
      Writeln(qwSize);                   {print file size}
    endif;
  FClose(hFile);                         {close the file}
```

FSize

```
endif  
end;
```

FSetPos**Declaration**

```
function FSetPos (var fi: tFile; qwPos: QWord): Boolean;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

qwPos

QWord. New position of the file pointer in bytes.

Return value

Boolean.

false in case of an invalid file handle of FSetPos failed.

true when successful.

Description

The function repositions the file pointer to the position provided with the qwPos variable.

Remarks

None

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
  i32Res:       Int32;
  qwPos:        QWord;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  SetQW(qwPos, 0,10);

  if FOpen(hFile,csFileNamePar,nenFMRead) then {open file with read access -
  error handling omitted}
  if FSetPos(hFile,qwPos) then {set file pointer}
    i32Res := BlockRead(hFile,bTest,-1); {read byte eleven}
  endif;
  Writeln(bTest); {print byte eleven}
  FClose(hFile); {close the file}
```

FSetPos

```
    endif  
end;
```

FGetPos**Declaration**

```
function FGetPos (var fi: tFile; var qwPos: QWord): Boolean;
```

Parameter

fi.

Referenz to tFile. File handle (see also FOpen).

qwPos

QWord. Retrieve file pointer position in bytes.

Return value

Boolean.

false in case of an invalid file handle or when FGetPos failed.

true when successful.

Description

The function retrieves the position of the file pointer. The position is return using the variable qwPos which is passed to the function as reference.

Remarks

None.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  bTest:        Byte;
  i32Res:        Int32;
  qwPos:        QWord;

begin
  csFileNamePar := "Test.bmp";           {set the file name}
  SetQW(qwPos, 0,10);

  if FOpen(hFile,csFileNamePar, nenFMRead) then {open file with read access -
  error handling omitted}
  if !(FSetPos(hFile,qwPos)) then          {set file pointer}
    Writeln(GetErrorMsg(GetError));       {print error message}
    return;
  endif;
```

```
if !(FGetPos(hFile,qwPos)) then           {retrieve file pointer
    Writeln(GetErrorMsg(GetError));       position}
    return;                               {print error message}
endif;
Writeln(qwPos);                           {print file pointer position}
FClose(hFile);                             {close the file}
endif
end;
```

6.3 Text files

Hint

You can use the functions described in chapter 6.2 also with text files. Especially the functions `FSetPos` and `FGetPos` are useful sometimes to position the file pointer when using text files.

Anyhow text files are usually accessed using lines contrary to binary files that are accessed using bytes.

TOpen

Declaration

```
function TOpen (var fi: tFile; csName: String; enFMode: tenFMode):  
Boolean; ifr;
```

Parameter

fi.

Referenz to tFile. File handle to a text file.

csName.

String. Name of the text file.

enFMode.

tenFMode. Desired access mode.

Mögliche Modi:

nenFMNone: Open without access rights.

nenFMRead: Open with read access.

nenFMUpdate: Open with read and write access. File pointer points to the beginning.

nenFMWrite: Open with write access. File pointer points to the beginning.

nenFMAppend: Open with write access. File pointer points to the end of the file.

Return value

Boolean.

false in case of an empty file name, an invalid Fmode or when TOpen failed.

true, when successful.

Description

The function opens a file as text file. If the specified file does not exist, it is created. In case no read access is specified. Attempts open a non existing file for reading returns false and an invalid file handle. The function returns true, when the file is opened successfully and false otherwise.

Remarks

The modifier ifr indicates that you can ignore the return value.

TOpen**Example**

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:         tFile;
begin
  csFileNamePar := "Test.txt";           {set the file name}
  if !TOpen(hFile,csFileNamePar,nenFMRead) then {open file with read access}
    Writeln(GetErrorMsg(GetError));      {print error message}
    return;
  endif;
  Writeln(hFile);                       {print file handle}
  if !TClose(hFile) then                 {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
end;
```

TClose**Declaration**

```
function TClose (var fi: tFile): Boolean; ifr;
```

Parameter

fi.

Referenz to `tFile`. File handle to a text file (see also `TOpen`).

Return value

Boolean.

True in case the file is closed successfully.

False in case of an invalid file handle or when `fClose` failed.

You can ignore the return value.

Description

The function closes a file. The return value is `true` when the file was closed successfully and `false` otherwise.

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  if !TOpen(hFile, csFileNamePar, nenFMRead) then {open file with read access}
    Writeln(GetErrorMsg(GetError));      {print error message}
    return;
  endif;
  Writeln(hFile);                       {print file handle}
  if !TClose(hFile) then                 {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif
end;
```

TEof**Declaration**

```
function TEof (var fi: tFile): Boolean;
```

Parameter

fi.

Referenz to `tFile`. File handle to a text file (see also `TOpen`).

Return value

Boolean.

`true`, in case the end of the file is reached, otherwise `false`.

Description

The function returns `false` until the end of the file is reached

Remarks

The function returns `false` in case of an invalid file handle. Thus testing the return value as exit criterion in a loop could result in an endless loop.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest:       String;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  if !TOpen(hFile,csFileNamePar,nenFMRead) then {open the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  while (TEof(hFile) = false) do        {loop until the end of the
                                        file is reached}
    csTest := TRead(hFile,1);           {read the next character byte}
    Writeln(csTest);                    {print the current character}
  endwhile
  TClose(hFile);                        {close the file}
end;
```

TFlush

Declaration

```
function TFlush (var fi: tFile): Boolean; ifr;
```

Parameter

fi.

Referenz to tFile. File handle to a text file (see also TOpen).

Return value

Boolean.

false in case of an invalid file handle or when TFlush failed

true, if successfully

Description

The function writes all internal buffered data to the file. Use this function when you opened a file with read/write access. This function assures that modified data is written to the file, so you can access the actual data.

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  csTest:       String;
  hFile:        tFile;
  qwPos:        QWord;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  SetQW(qwPos, 0, 0);

  if !TOpen(hFile, csFileNamePar, nenFMUpdate) then {open file with read/write
                                                    access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  TWriteStr(hFile, "A");                 {write a character}
  if !(TFlush(hFile)) then              {save internal buffer to
                                        disk}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
```

TFlush

```
if !(FSetPos(hFile,qwPos)) then
    Writeln(GetErrorMsg(GetError));           {print error message}
    return;
endif;
csTest := TRead(hFile,1);                   {read a character}
Writeln(csTest);
if !TClose(hFile) then                      {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
endif;
end;
```

TRead**Declaration**

```
function TRead (var fi: tFile; xoCount: tSize): String;
```

Parameter

fi.

Referenz to tFile. File handle to a text file (see also TOpen).

xoCount.

Data type tSize equals Int32. Number of characters to read. When xoCount is bigger than the number of characters in the file, the function reads all characters until EOF (end of file) is reached.

Return value

String.

Read data. An empty string and an error code is returned in case of an invalid file handle, xoCount <= 0 or when TRead failed.

Description

The function reads xoCount characters from the text file and returns the characters as a string. When xoCount is bigger than the number of characters in the file, the function reads all characters until EOF (end of file) is reached. An empty string and an error code is returned in case of an invalid file handle, xoCount <= 0 or when TRead failed.

Remarks

LF resp. CR+LF (on Windows machines also Ctrl-Z='\x1A') are truncated. See also nFF*Mask (nFFPresNLMask etc.).

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:         tFile;
  csTest:        String;
begin
  csFileNamePar := "Test.txt";           {set the file name}
  if !TOpen(hFile, csFileNamePar, nenFMRead) then {open file with read access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  csTest := TRead(hFile, 4);             {read 4 characters}
```

TRead

<pre> Writeln(csTest); TClose(hFile); endif</pre>	<pre>{print the read characters} {close the file}</pre>
--	---

TReadln

Declaration

```
function TReadln (var fi: tFile): String;
```

Parameter

fi.

Referenz to tFile. File handle to a text file (see also TOpen).

Return value

String.

Read data. An empty string and an error code is returned in case of an invalid file handle or when TReadln failed.

Description

The function reads all character until an end of line or and EOF (end of file) is detected. The read data is returned as a string and the file pointer is positioned behind the last read character, i.e. the file pointer usually points to the next line. An empty string and an error code is returned in case of an invalid file handle or when TReadln failed.

Remarks

LF resp. CR+LF are truncated (on Windows machines also Ctrl-Z='x1A'). See also nFF*Mask (nFFPresNLMask etc.).

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest1:      String;
  csTest2:      String;
begin
  csFileNamePar := "Test.txt";           {set the file name}
  if !TOpen(hFile,csFileNamePar,nenFMRead) then {open file with read access}
    Writeln(GetErrorMsg(GetError));     {print error message}
    return;
  endif;
  csTest1 := TReadln(hFile);           {read first line}
  csTest2 := TReadln(hFile);           {read second line}
  Writeln(csTest1);                    {print first line}
  Writeln(csTest2);                    {print second line}
  TClose(hFile);                       {close the file}
end;
```

TWriteStr

Functions

```
function TWriteStr (var fi: tFile; cs: String): ?t31Bit; ifr;  
function TWriteCh (var fi: tFile; c: Char): ?t31Bit; ifr;  
function TWriteI32 (var fi: tFile; i32: Int32): ?t31Bit; ifr;  
function TWriteDW (var fi: tFile; dw: DWord): ?t31Bit; ifr;  
function TWriteBo (var fi: tFile; bo: Boolean): ?t31Bit; ifr;  
function TWriteRe (var fi: tFile; r64: Real64): ?t31Bit; ifr;  
function TWritePtr (var fi: tFile; pv: Pointer): ?t31Bit; ifr;
```

Parameter

fi.

Referenz to `tFile`. File handle to a text file (see also `TOpen`).

cs, c, i32, dw, bo, r64, pv.

`String`, `Char`, `Int32`, `DWord`, `Boolean`, `Real64`, `Pointer` (According the the function – see function declarations above).

Data to be written to a text file.

Return value

?t31Bit.

Number of written characters.

Description

The function writes the data provided with the second parameter to the file. The function returns the number of characters written. Following calls to the function simply append the data. You should use the function `WriteLn` in order to get a line feed after writing the data.

Remarks

The function `TWritePtr` writes the address to the text file prepending the address operator.

The modifier `ifr` indicates that you can ignore the return value.

TWriteStr**Example**

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest1:      String;
  csTest2:      String;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  csTest1 := "Text1";
  csTest2 := "Text2";
  if !TOpen(hFile,csFileNamePar,nenFMWrite) then {open file with write
                                                access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  if !(TWriteStr(hFile, csTest1)) then      {write text to file}
    Writeln(GetErrorMsg(GetError));      {print error message}
    return;
  endif;

  TWriteStr(hFile,csTest2);              {write text to file
- error handling omitted}

  if !TClose(hFile) then                  {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif
end;
```

TWriteInStr

Functions

```
function TWriteInStr (var fi: tFile; cs: String): ?t31Bit; ifr;  
function TWriteInCh (var fi: tFile; c: Char): ?t31Bit; ifr;  
function TWriteInI32 (var fi: tFile; i32: Int32): ?t31Bit; ifr;  
function TWriteInDW (var fi: tFile; dw: DWord): ?t31Bit; ifr;  
function TWriteInBo (var fi: tFile; bo: Boolean): ?t31Bit; ifr;  
function TWriteInRe (var fi: tFile; r64: Real64): ?t31Bit; ifr;  
function TWriteInPtr (var fi: tFile; pv: Pointer): ?t31Bit; ifr;
```

Parameter

fi.

Referenz to tFile. File handle to a text file (see also TOpen).

cs, c, i32, dw, bo, r64, pv.

String, Char, Int32, DWord, Boolean, Real64, Pointer (According the the function – see function declarations above).

Data to be written to a text file.

Return value

?t31Bit.

Number of written characters.

Description

The function writes the data provided with the second parameter to the file. The function returns the number of characters written. After the data is written to the file an additional new line is written to the file. To write to a file without appending an new line automatically please use the function WriteIn.

Remarks

The function TWritePtr writes the address to the text file prepending the address operator.

The modifier ifr indicates that you can ignore the return value.

TWriteLnStr**Example**

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest1:      String;
  csTest2:      String;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  csTest1 := "Zeile 1";
  csTest2 := "Zeile 2";

  if !TOpen(hFile,csFileNamePar,nenFMWrite) then {open file with read access}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
  TWriteLnStr(hFile, csTest1);           {print first line
                                         Error handling omitted}
  TWriteLnStr(hFile,csTest2);           {print second line
                                         Error handling omitted}
  if !TClose(hFile) then                 {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
  endif
end;
```

Write**Declaration**

```
procedure Write (fi tFile (optional), outputParameter (optional));
```

Parameter

Optional.

fi.

tFile. File handle to a text file (see also TOpen).

Optional.

outputParameter.

Any constant or variable to be written to the file.

Return value

None.

Description

The passed constant resp. Variable is written to a file if a valid file pointer is passed. The output is directed to standard output (console) if no file pointer is passed nor StdOut is specified. Take care when writing to the console. No line feed is performed when the end of the line is reached instead the previous written data is overwritten. To write into the next line use the function Writeln.

Remarks

Both parameters are optional.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest:       String;
  i32Test:      Int32;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  csTest := "Zeile 1";
  i32Test := 10;

  if !TOpen(hFile,csFileNamePar,nenFMWrite) then {open file with write
                                                    access}
```

Write

```
        Writeln(GetErrorMsg(GetError));
        return;
    endif;
    Write(hFile, csTest);           {write the first variable
                                   to the file}
    Write(hFile, i32Test);        {write the second variable
                                   to the file}
    Write(csTest);               {write the first variable
                                   to the console}
    Write(i32Test);              {write the second variable
                                   to the console}
    Write(StdOut, csTest);       {write the first variable
                                   to standard out}
    Write(StdOut, i32Test);      {write the second variable
                                   to standard out}
    if !TClose(hFile) then
        Writeln(GetErrorMsg(GetError));
        return;
    endif;
end;
```

Writeln**Declaration**

```
procedure Writeln (fi tFile (optional), outputParameter (optional));
```

Parameter

Optional

fi.

tFile. File handle to a text file (see also TOpen).

Optional.

outputParameter.

Any constant or variable to be written to the file.

Return value

None.

Description

The constant resp. Variable is written to a new line in the file if a valid file pointer is passed. To append a parameter in the same line use the function Write instead. In case no file is passed nor is StdOut specified as first parameter, the output is directed to the standart output (console).

Remarks

Both parameters are optional.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
  csTest:       String;
  i32Test:      Int32;

begin
  csFileNamePar := "Test.txt";           {set the file name}
  csTest := "Zeile 1";
  i32Test := 10;

  if !TOpen(hFile,csFileNamePar,nenFMWrite) then {open the file with write
                                                access}
    Writeln(GetErrorMsg(GetError));
```

Writeln

```
    return;
endif;
Writeln(hFile, csTest);           {write the first variable
                                  to the file}
Writeln(hFile, i32Test);         {write the second variable
                                  to the file}
Writeln(csTest);                 {write the first variable
                                  to the console}
Writeln(i32Test);                {write the second variable
                                  to the console}
Writeln(StdOut, csTest);         {write the first variable
                                  to standard out}
Writeln(StdOut, i32Test);        {write the second variable
                                  to standard out}
if !TClose(hFile) then           {close the file}
    Writeln(GetErrorMsg(GetError));
    return;
endif
end;
```

7 POOL base classes

Taken from the pool.pli library.

7.1 The base class toRoot

Introduction

toRoot is the base class to all classes in POOL. Omitting a parent class when declaring a new class implies that the new class is inherited from POOL's base class toRoot.

The constructor and destructor are the most important elements of the base class toRoot. Both have to be called from the corresponding method of the inheriting class.

The second part of the pool tutorial provides a detailed description on inheritance.

In this chapter we also describe the toEMRoot class. toEMRoot is a direct or indirect parent class to all POOL event and POOL mutex classes. toEMRoot inherits also from toRoot like all other POOL classes do.

Remarks:

Most of the description of the base class toRoot is taken from the pool.pli library.

The class toRoot

Definition

```

type
  toRoot = object
    constructor poInit;           {initialize private elements}
    constructor poReinit;        {reinitialisation (result is nil
                                if no or an invalid type is listed in
                                pstVOT)}
    destructor vDone; virtual;   {free allocated resources}

    procedure vWriteObj (var fiOut: tFile; stOpt: tstOptions); virtual;
    {standard output function: All elemente are put out}
    {Attention: You should not use any HWrite function in conjunction with
     StdOut, whereas Write and Writeln are okay
     (they include special treatment)!}
end; {toRoot}
tpoRoot = ^toRoot;

```

toRoot**Pointer to the object**

`tpoRoot`.

Parent class

None . `toRoot` is the parent class of all other POOL classes.

Description

`toRoot` is the direct or indirect base class to all POOL classes. `toRoot` is added as parent class to all other classes (except to the classes in `pool.lib`) to provide the possibility to call the constructor of the parent class, and to ensure the existence of a standard destructor and a standard output mechanism.

Remarks

Please see the remarks of the methods `poInit`, `poReinit`, and `vDone` below.

Methods

constructor `poInit`;

Constructor to initialize private elements.

constructor `poReinit`;

Reinitialisation (Result is `nil`, in case that no type or the wrong type is registered in `pstVOT`).

destructor `vDone`; `virtual`;

Deletes the data structure.

procedure `vWriteObj` (`var fiOut: tFile`; `stOpt: tstOptions`); `virtual`;

Standard output method. Puts out all elements.

toRoot**Method declaration**

```
constructor polnit;
```

Parameter

None.

Return value

`poEvent`.

Pointer to the object or `nil` in case of an error.

Description

The constructor initializes the private elements. You have to ensure to call the constructor of the parent class when inheriting. That is you have to call the constructor of the base class `poRoot`, when you do not specify a parent class. For details please see part two of this tutorial.

Remarks

(Source: `pool.pli`)

You can ignore the return value of a constructor call when working with existing objects like static variables and the only possible error is that the system runs out of memory. However the result of an constructor call of a parent class has to be handled.

In a constructor that does not produce errors on his own, that is errors occur only within inherited constructors, it is neither allowed nor necessary to call a destructor. Calling a destructor could result into an avalanche of useless destructor calls.

A constructor that produces own errors, a destructor call is necessary to clean up the parts which have been initialized so far. The clean up should be performed by the destructor of the class.

Constructor calls for `oSelf` (z.B. `inherited polnit` or `toRoot.polnit`) are allowed only within the constructor itself. Constructor call using Type-Casts (e.g. `tpoRoot(@oSelf)^.polnit`) are allowed, but are useful only in some special cases, since the casted type is inserted into the object instead of the real type.

Constructor call for `oSelf` specifying an object type (e.g. `toRoot.polnit`) are reasonable only if you explicitly want to skip constructors of ancestors. Be careful since you have to do some corrections manually when you change the hierachy of the objects!

`const` or `static` objectes are already initialized and can be used without further initialization. Whereby only fields are initialized (inclusive the hidden VOT field), but no conctructor is called! These types of objects are useful mostly in simple cases.

toRoot

You have to call a constructor before you can use an uninitialized object. After having finished using the object you have to deinitialize the object with a destructor call! In POOL all data is initialized with zero, and the destructor also clears the data. This is the reason why the constructor expects an object with an empty data area. If the data is not deleted, as is the case with multiple constructor calls without destructor calls, `nil` is returned, `EINVAL` is returned as error code, and the object is not modified (at least when all constructors are implemented correctly, that is, modification of the object's data takes place only after a successful call of the parent constructor). In case that the data area of an object is used alternatively e.g. when the object is placed in the variant part of a structure, then the application has to delete the overlapping fields explicitly before the constructor is called.

Local objects (except the ones declared static) can only be used by the thread they belong to. Especially initialization is prohibited by another thread e.g. by passing the objects address via a global variable and calling the constructor from another thread.

Example

```
{Calling the parent class constructor within the constructor of the
inheriting class }

if inherited poInit = nil then {calling the constructor of the parent class
                               (in this case toRoot) failed?}
    poInit := nil;              {return value of the constructor = nil}
    return;                     {exit the constructor}
endif;
```

toRoot**Method declaration**

```
constructor poReinit;
```

Parameter

None.

Return value

poRoot.

Pointer to the object or `nil`, in case that no type or the wrong type is registered in `pstVOT`).

Description

The `poReinit` constructor is used to reinitialize the dynamic elements of already initialized object. Calling `poReinit` is allowed only on objects that were initialized by a previous call of a constructor before (e.g. the ones in the `static` segment). Mutex and events are not allowed as initialized constants, although the compiler can not detect it.

`poReinit` is defined as constructor, since this is the only way to pass the parameter that are needed for internal examination. Never the less it is not a real constructor and therefore can never be used with `new` key word! You are not allowed to return a value other than `nil` in your `poReinit` constructor if the object is not initialized correctly. `Reinit` can be used to implement a constructor (e.g. `Init`), that allows to work with both, new objects and already initialized objects. This is done by calling an inherited `Reinit`. The object was previously initialized if `Reinit` was successful, otherwise you have to call an inherited `Init` to initialize the object.

Remarks

None

toRoot**Method declaration**

```
destructor vDone; virtual;
```

Parameter

None.

Return value

None.

Description

The destructor frees the memory during object deletion.

Remarks

The standard destructor vDone is the first virtual method of all objects. You can call vDone also on uninitialized or deleted objects (with no actions performed of course). This eases cleaning up e.g. in vDeinit.

For a more detailed description on destructors see tutorial part 2.

Example

```
{call the destructor of the parent class from the inheriting classes  
destructor}  
  
inherited vDone; {calling the destructor of the parent class}
```

toRoot**Method declaration**

```
procedure vWriteObj (var fiOut: tFile; stOpt: tstOptions); virtual;
```

Parameter

fiOut.

tFile as reference. Output file or StdOut.

stOpt.

tstOptions. Output options.(see also chapter 18.3).

Return value

None.

Description

The function puts out all elements to output file resp. standard output.

Remarks

You should not use any HWrite function in conjunction with StdOut, whereas Write and Writeln are okay since the include special treatment.

7.2 Object toEMRoot

Object toEMRoot

Definition

```

type
  toEMRoot = object (toRoot)
  private
    ?pNext:          Pointer;    {pointer for linking of event lists}
    ?pPrev:          Pointer;
    ?pNext:          Pointer;    {pointer for linking of AddEvent}
    ?unTypeFlags:   tunWord;    {type and divers internal flags}
    ?wThreadID:     Word;       {ThreadID for Mutex}

  public
    constructor poInit;          {initialization of private elements}
    destructor vDone; virtual;  {data deletion and memory freeing if needed}
    {Attention: Destructors of events are not allowed to modify or delete }
    {data that is still used by other tasks or threads (including POOLs}
    {runtime system)!}
  end; {toEMRoot}
  tpoEMRoot = ^toEMRoot;

```

Pointer to Object

tpoEMRoot.

Parent object

toRoot. Object, whose properties and methods are inherited.

Description

toEMRoot is a parent class to all POOL-Event and POOL-Mutex objects.

Remarks

None.

Methods

constructor poInit;

Initializes private elements. Calls the constructor of the base class poRoot.

toEMRoot

destructor `vDone`; `virtual`;

Deletes the data and if necessary frees the memory. Calls the destructor of the base class `poRoot`.

toEMRoot**Method declaration**

```
constructor poInit;
```

Parameter

None.

Return value

poEvent.

Pointe to the object; nil in case of an error.

Description

The constructor is used to initialize the private elements. Additional the constructor of the base class is called.

Remarks

None.

Example

```
oEvent.poInit; {Calling the constructor on the object oEvent}
```

toEMRoot**Method declaration**

```
destructor vDone; virtual;
```

Parameter

None.

Return value

None.

Description

Deletes the data and frees the memory. Additionally the destructor of the base class `poRoot` is called.

Remarks

Attention: Destructors of events are not allowed to modify or delete data that is still used by other tasks or threads (including POOLs runtime system)!

Example

```
oEvent.vDone; {calling the destructor on the object oEvent}
```

8 Event handling

Taken from the pool.pli library.

8.1 Introduction

Using events makes it easier for the software developer to react to internal and external events. Timer and entry events are most often used event types.

Timer events are always triggered when a timer flows over, in other words if the set time has expired. By using timer events that are triggered cyclically it is possible to implement a time reference, which makes it possible to process statements in regular intervals. Applications like these can often be found in control engineering for instance.

Keyboard events are needed to be able to react to entries made by the user. The big advantage of events becomes apparent in this case. Without an event system the individual enter keys would have to be polled regularly by the program in order to find out whether a key is being operated. Frequent polling would be necessary to avoid that an entry is "overlooked." Polling leads to an unnecessarily high runtime of the program. The poll would have to be done even in a state of rest, during which the system only waits for the occurrence of events. In the worst case imaginable there would be only little CPU time left for processing other program parts. There is no polling with the event system, since the operation of a key triggers an event, which "only" has to be processed within appropriate routines. It will not be of interest to us at this point how the operating system recognizes these events and how the events are passed to the application.

Events can even be signaled using special functions, if they did not occur. This is used for example to end threads from the main program.

A distinction is made between two different function types of event functions:

1. Wait Functions:

With wait functions the system waits for the occurrence of events. The program is put into a sleep mode, in which it uses up no processing time. It is the event that causes the program to exit the sleep mode and to continue.

2. NoWait procedures

NoWait procedures continue immediately without waiting for the occurrence of events. In contrast to the wait functions it is therefore possible to process several events at the same time. There are the WaitEvent (one event) and WaitEvents functions (several events) that are used to wait for the occurrence of one or any number of events. When WaitEvent(s) is called, the program is put into sleep mode. If an event occurs the

program wakes up determines the type of event to activate the appropriate processing routine. A NoWait-Event object, which also includes the return value of the event function, is assigned to each event. This object is passed as parameter to the NoWait function during the function call. Since we are dealing with an object, it has to be initialized before it can be used via the constructor and then deleted again using a destructor call.

If several events occur simultaneously, the first event in the transmission list of WaitEvents is evaluated first. This makes it possible to assign priorities.

If it is necessary to wake the program in regular intervals and to "force" a continuation after an elapsed time, then this can be done by using timer events. If such an event is passed last in WaitEvents, then it has the lowest priority and is only executed, if there is no other event.

Important:

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, or the appropriate NoWait versions).

Please see the following sections for a detailed description of the individual objects and functions.

8.2 Event objects

8.2.1 Object toEvent

```
{toEvent: General POOL-Event object and ancestor of all other Event objects}
type
  toEvent = object (toEMRoot)
    dwErrNo:      DWord;           {optional error # as return value}
    pstValType:  tpstType;         {describes type of unVal)}
    unVal:       tunEventVal;     {optional return value}
    constructor  poInit;          {initialization of privat elements}
    function     boQueued: Boolean; {whether event is in an internal queue}
    procedure    vCompletion; virtual; { (name reserved for a not yet existing}
                                           {Completion function)}
    procedure    vSignal;         {signal an event }
    procedure    vLock;           {lock an event, not implemented yet}
    function     boTryLock: Boolean; {try to lock an event not impl.}
    procedure    vUnlock;         {unlock an event not impl.}
    {Remark: As long as vLock and vUnlock are not implemented, additional
     Mutex and Event objects have to be used for synchronisation if needed!}

  end; {toEvent}
  tpoEvent = ^toEvent;
  tapoEvent = array[0..] of tpoEvent;
  tpapoEvent = ^tapoEvent;
```

Pointer to the object

tpoEvent.

Array of pointer to the object

tapoEvent = array[0..] of tpoEvent.

tpapoEvent = ^tapoEvent;

Parent object

toEMRoot.

Description

toEvent is the base class to all POOL event classes. All other event classes have to inherit directly or indirectly from toEvent.

toEvent**Remarks**

vLock, boTryLock, and vUnlock are currently not implemented. Therefore you have to use additional mutex and event objects to synchronize concurrent access if necessary.

Methods

constructor polnit;

Standard constructor to initialize the private elements of the parent class.

function boQueued: Boolean;

Query whether the event is in an internal queue.

procedure vCompletion; virtual;

Reserving the name vCompletion for future use.

procedure vSignal;

Signal event.

procedure vLock;

Lock event. Not implemented yet.

function boTryLock: Boolean;

Try to lock an event. Not implemented yet.

procedure vUnlock;

Unlock event. Not implemented yet.

toEvent**Public properties****Property**

dwErrNo

Data type

DWord.

Description

Optional error code. You can use it for instance to process an error occurred during a NoWait function (for detail see the description of the appropriate function).

Property

pstValType.

Data type

tpstType.

Description

Type description for unVal (currently only used as place holder).

Property

unVal

Data type

tunEventVal

Consists of dw (DWord), i32 (Int32) and pv (Pointer).

Description

Optional return value. This property is used to return the result of calls to functions that have an oNWEvent parameter.

toEvent**Method declaration**

```
constructor poInit;
```

Parameter

None.

Return value

```
tpoNWEvent.
```

Pointer to the event object or `nil` in case of an error.

Description

Initialisation of private elements of the parent class (calls the constructor of the parent class `toEMRoot`).

Remarks

None.

Example

```
{initialisation}
begin
  oEvent.poInit;   {initialisation of an event object}
end.
```

toEvent**Method declaration**

```
function boQueued: Boolean;
```

Parameter

None.

Return value

Boolean.

`true` if the event is signaled either manually by calling `vSignal` or when the preset time is elapsed, otherwise `false` is returned.

Description

The function indicates whether the event was added to an internal queue. To avoid system threads from being killed you can wait for events in `Deinit` that are not signaled yet. However you have to ensure that the events get signaled within reasonable time.

Remarks

None.

Example

```
{Within deinitialisation}
while oTEvent.boQueued do {Loop until event to kill thread is signaled}
  Wait(25);
endwhile;
DestroyThread(hTHandle);
oTEvent.vDone;
```

toTEvent**Method declaration**

```
procedure vSignal;
```

Parameter

None.

Return value

None.

Description

Signals an event. You can retrieve signaled events using the function `GetEventSignaled`. A typical application is to terminate threads from within the main program.

Remarks

None.

Example

```
{In a Thread}

repeat
  if(WaitEvent(oTEvent) = nil) then {waiting for a timer event }
    Writeln(GetErrorMsg(GetError)); {print error message}
    return;
  endif;
  Writeln("Event occured");          {print after each occurred event}
until GetEventSignaled;             {As long as the event is not signaled -
                                     a timer event does not signal the event}

{Within deinitialisation:}

oTEvent.vSignal;                    {Kills the thread - GetEventSignaled
                                     becomes true}
```

toTEvent**8.2.2 Object toTEvent****Object toTEvent****Definition**

```

type
  tpoTEvent = ^toTEvent;
  toTEvent = object (toEvent)
  private
    ?pstNextTEvt: tpoTEvent;           {link for timer thread}
    ?i32MSec:     Int32;                 {delta-t in ms}
    ?dwTAbs:      DWord;                 {abs. Zeit für das Event}

  public
    constructor poInit(i32DTms: Int32); {initialisation of private elementes}
    procedure   vSetDT(i32DTms: Int32); {set delta-t in ms}
    procedure   vSetRM(boRM: Boolean);  {set/reset retrigger mode}
  end;

```

Pointer to the object

tpoTEvent.

Parent object

toEvent.

Description

The object `toTEvent` provides timer with an arbitrary time span in ms. The timer gets signaled when the preset time is elapsed. You can also set the timer to a retriggered mode, i.e. the timer starts over again when the time is elapsed.

Remarks

The timer is triggered when the function `WaitEvent(s)` is called. The elapsed time belongs to the timer object even if multiple POOL threads wait for the same timer object. That is, the time which is set to the timer is , for a logical point of view, used to signal the timer and not the `WaitEvent(s)` functions. In case only one POOL thread is waiting for the object you can consider it as timeout for the `WaitEvent(s)` function.

The timer is started during the next `WaitEvent(s)`, only if the result of the previous timeout was returned.

toTEvent**Public properties**

None.

You should use the methods to access the properties.

Methods

```
constructor polnit(i32DTms: Int32);
```

Constructor to initialize private elements.

```
procedure vSetDT(i32DTms: Int32);
```

Used to preset a delta-t in ms.

```
procedure vSetRM(boRM: Boolean);
```

Used to enable/disable retrigger mode.

toTEvent**Method declaration**

```
constructor poInit(i32DTms: Int32);
```

Parameter

i32DTms.

Int32. Time in ms between timer start and raising timer event.

Return value

tpoTEvent.

Pointer to timer event object or nil in case of an error.

Description

The method initializes a timer object and sets the time in ms. Additionally the constructor calls the constructor of the parent class toEvent.

Remarks

None.

Example

```
oTEvent1.poInit(1000); {initialize the timer object setting a time of 1000ms}
```

toTEvent**Method declaration**

```
procedure vSetDT(i32DTms: Int32);
```

Parameter

i32DTms.

Int32. Time in milli seconds between the timer start and the signaled event.

Return value

None.

Description

The function sets the timer to i32DTms in milli seconds. You can use the function to change the timer to a different value than the one that was set during initialization.

Remarks

The function must not be interrupted by a WaitEvent of the same event. Normally vSetDT is called before the thread, that is waiting for the event, is started. This ensures the condition automatically.

Example

```
oTEvent1.vSetDT(100); {set the timer to 100ms}
```

toTEvent**Method declaration**

```
procedure vSetRM(boRM: Boolean);
```

Parameter

boRM.

Boolean.

Set whether the retrigger mode is active or not.

true = turn retrigger mode on.

false = turn retrigger mode off.

Return value

None.

Description

The retrigger mode is activated if the function is called with boRM = true. That is the timer starts over again if the preset time has elapsed. This allows to create precise periodical events provided you take care not to signal the timer event using vSignal and not to run the system with high load. It is possible to miss an event if the system load is to high.

Remarks

The internal timeout state is deleted when the function vSignal of the parent class is called.

The function SetRM must not be interrupted by a WaitEvent of the same event.

Example

```
module Test;

private
var
  oTEvent1: toTEvent;
  oTEvent2: toTEvent;

procedure vMain;
begin
  repeat
    {oTEvent1 is handled first if both events occur simultaneously}
    if WaitEvents(oTEvent1,oTEvent2) = tpoEvent(@oTEvent1) then
      Writeln("Timer1 event");      {print a hint}
    else
```

toNWEvent

```
        Writeln("Timer2 event");           {print a hint}
    endif;
    until KeyPressed <> 0;                 {exit if a key is pressed}
end;

{vDeinit: deinitialize module}
procedure vDeinit;                       {Deinitialize the event object}
begin
    oTEvent1.vDone;
    oTEvent2.vDone;
end;

begin
    oTEvent1.poInit(1000);                {set timer1 to 1000ms}
    oTEvent2.poInit(2000);                {set timer2 to 2000ms}

    oTEvent1.vSetRM(true);                {set timer1 to retrigger mode}
    oTEvent2.vSetRM(true);                {set timer2 to retrigger mode}
end.
```

toNWEvent**8.2.3 Object toNWEvent**

```
type
  toNWEvent = object (toEvent)
    private
      ?pOrder:      Pointer;      {pointer to Working-Thread-Job}

    public
      constructor poInit;          {initialize private elements}
end;
tpoNWEvent = ^toNWEvent;
```

Pointer to the object

tpoNWEvent.

Parent object

toEvent.

Description

NoWait events are event which are not waited for (see also introduction). That is you pass a NoWait event to a NoWait procedure and the procedure does not wait for the event instead the program continues executing. The return value is written to the oNWEvent object and can be retrieved later on. You have to use the WaitEvent(s) functions to wait for the events. After the event is signaled you can retrieve the information from the oNWEvent object. See the description of the parent object toEvent for the information you can get (public properties) from the object.

Remarks

Use the destructor of the parent class toEvent to delete the object.

You should not delete unsignaled events using the destructor since this would terminate system internal threads and some resources would not be freed until the program terminates.

Methods

constructor poInit;

Initializes private elements of the parent class.

toNWEvent**Method declaration**

```
constructor polnit;
```

Parameter

None.

Return value

`tpoNWEvent`.

Pointer to the object or `nil` in case of an error. The constructor also calls the constructor of the parent class `toEvent`.

Description

The method initializes a NWEvent object.

Remarks

None.

Example

(see `toTEvent`)

Anmerkung:

The destructor of the parent class is called in `vDone`.

Wait

8.3 Event functions

Anmerkung:

Events, issued by the Commander are described in chapter 9.

Declaration

```
procedure Wait (i32Tms: Int32);
```

Parameter

i32Tms.

Int32. Time to wait.

Return value

None.

Description

The procedure waits until the time passed via i32Tms in milli seconds has elapsed. Program execution stops at the location where the function is called and the thread is set into sleep mode until the event is signaled.

Remarks

None.

Example

```
repeat                {loop}
  Wait(30);           {wait for 30ms}
until KeyPressed <> 0; {exit loop when a key is pressed}
```

WaitEvent

Declaration

```
function WaitEvent(var oEvent: tOEvent): tPOEvent;
```

Parameter

oEvent.

tOEvent. The event to wait for.

Return value

tPOEvent.

Pointer to the signaled event. `nil` is returned in case of an error and an error code is set. Possible errors are invalid parameters and uninitialized events.

Description

The function waits for the passed event. Program execution stops at the location where the function is called and the thread is set into sleep mode until the event is signaled. A pointer to the event object is return if successful or `nil` otherwise.

Remarks

None.

Example

(see also `GetEventSignaled`)

```
procedure vThread1;
begin
  repeat
    Writeln("Thread1");
    if (WaitEvent(oTEvent1)= nil then
      Writeln(GetErrorMsg(GetError));
    return;
  endif;
  until GetEventSignaled;
end;
```

{loop}
{print a text}
{wait for a timer event}
{is continued each time the timeout}
{occurs (each 30ms in this case)}

{the loop is exited when the event is signaled in the main thread. The timer event does not exit the loop.!}

WaitEvents

Declaration

```
function WaitEvents(..): tpoNWEvent;
```

Parameter

```
..
```

```
toEvent.
```

A list of events to wait for. You can also pass pointer to events and event arrays.

Return value

```
tpoNWEvent.
```

Pointer to the first event in the (ordered) list that is signaled even if two or more events are signaled. The order of event occurrence is irrelevant. `nil` is returned in case of an error and an error code is set in the event object. Possible errors are invalid parameters and uninitialized events.

Description

The function waits until an event occurs.

The calling thread stops execution at the point where the `WaitEvents` function is called and enters "sleep mode" until an event is signaled. A pointer to the first signaled event with respect to the order in the parameter list is returned even if more events are signaled. The order of event occurrence is irrelevant! Since the returned event is set to not signaled a subsequent call to `WaitEvents` function returns the next signaled event.

Remarks

None.

Example

(see also `GetEventSignaled`)

Note: The example program has no practical use. Its sole purpose is to illustrate the different priorities of evaluation. The function result is used to determine the event. The description of the function `GetEventNo` includes an example that shows how to use the function result in real applications. Error handling is omitted in this example.

```
module Test;

private
  var
    oNWEvent1: tpoNWEvent;
    oNWEvent2: tpoNWEvent;
```

WaitEvents

```
procedure vMain;
begin
  oNWEvent2.vSignal;
  oNWEvent1.vSignal;

  // oNWEvent1.vSignal; {modifying the order of the event does not modify
  // oNWEvent2.vSignal; the result of WaitEvents}

  {evaluate the events - error handling is omitted (see WaitEvents)}
  if WaitEvents(oNWEvent1,oNWEvent2) = tpoEvent(@oNWEvent1) then
    Writeln("NWEvent1"); {is executed}
  else
    Writeln("NWEvent2");
  endif;

  {evaluate the events again - usually you would use a loop for evaluation}
  if WaitEvents(oNWEvent1,oNWEvent2) = tpoEvent(@oNWEvent1) then
    Writeln("NWEvent1");
  else
    Writeln("NWEvent2"); {is executed since NWEvent1 was handled before}
  endif;
end;

procedure vDeinit;
begin
  oNWEvent1.vDone;
  oNWEvent2.vDone;
end;

begin
  oNWEvent1.poInit;
  oNWEvent2.poInit;
end.
```

GetEventNo**Declaration**

```
function GetEventNo: ?t15Bit;
```

Parameter

None.

Return value

?t15Bit.

Number of the signaled event that was evaluated last via WaitEvent(s).

Description

The function returns the number of the event that was returned during the last WaitEvent(s) function of the current POOL thread (see also WaitEvent(s)). 0 is returned in case of an error.

You can use GetEventNo evaluate the event using a case statement.

Remarks

The function WaitEvents numbers all passed event parameters including empty parameters and also parameters within an array to determine the event number.

Some library functions (Wait and all functions that have a NoWait counterpart) use WaitEvent(s) internal and thus modify the internal value like the last evaluated event. As of this reason you should call and evaluate GetEventNo directly after the call to WaitEvent(s). If you can not evaluate the result immediately you can also buffer the event number in a variable.

Example

(see also GetEventSignaled)

```
module Test;

private
var
  oTEvent1: toTEvent;
  oTEvent2: toTEvent;

procedure vMain;
begin
  repeat
    if (WaitEvents(oTEvent1,oTEvent2) = nil) then
      Writeln(GetErrorMsg(GetError));
    return;
  endif;
```

GetEventNo

```
case GetEventNo of    {retrieves the number of the first active event in
                      the list}

  1: {is always executed when timer1 event is signaled }
    Writeln("Timer1 event signaled");

  2: {is executed when timer2 event is signaled and timer1 event is not
      signaled resp. timer1 event was handled in the previous loop and
      therefore was reset to not signaled}
    Writeln("Timer2 event signaled");

endcase;

until KeyPressed <> 0; {exit condition}
end;

procedure vDeinit;
begin
  oTEvent1.vDone;
  oTEvent2.vDone;
end;

begin
  oTEvent1.poInit(1000); {initialize timer1 event}
  oTEvent2.poInit(2000); {initialize timer2 event}
  oTEvent1.vSetRM(true); {execute timer1 event in retrigger mode}
  oTEvent2.vSetRM(true); {execute timer2 event in retrigger mode}
end.
```

GetEventSignaled

Declaration

```
function GetEventSignaled: Boolean;
```

Parameter

None.

Return value

Boolean.

`true`. An event is signaled and evaluated.

`false`. If an error occurred during the `WaitEvent(s)` call or in case the event is a timer event.

Description

The function retrieves the state of the last active and evaluated event of the calling thread i.e the event that woke up the thread. The `WaitEvent(s)` function returns this event as function result (see `WaitEvents` in chapter 8.3).

The function returns `true` if the event is signaled or `false` when an error occurred during a `WaitEvent(s)` call or in case of a timer event.

Remarks

None

Example

(see also chapter 11 on threads)

```
module Test;

procedure vThread1;
procedure vThread2;
procedure vCreateThread;

private
var
  hTHandle1: tTHandle;
  oTEvent1: toTEvent;

{Quellcode des Threads}
procedure vThread1;
begin
  repeat
    Writeln("Timer expired");           {is executed after a timer event (and
                                         during the first run of the thread,
```

GetEventSignaled

```

                                since the loop is controlled at the end)}
    if (WaitEvent(oTEvent1) = nil) then
        Writeln(GetErrorMsg(GetError));
        return;
    endif;
until GetEventSignaled;          {true, if the event is not a timer event
                                i.e. signaled within main using vSignal}
Writeln("exit thread1 ");        {after the event has been signaled}
end;

procedure vCreateThread;
begin
    hTHandle1 := CreateThread(0);  {create the first thread}
    {the thread is up and running from here on}
    if(hTHandle1 = nInvHandle) then {in case of an error}
        Writeln("Error thread 1-",GetErrorMsg(GetError));
        {if current thread = thread1}
    elseif (hTHandle1 = GetThreadHandle) then
        vThread1;                  {call thread's procedure vThread1}
        return;                    {exit thread}
    endif

    repeat                          {continue running thread}
        Wait(30);
    until KeyPressed<>0;            {terminate program when key was hitted}
end;

procedure vMain;
begin
    vCreateThread;                  {call procedure to initialize threads}
end;

{deinitialisation}
procedure vDeinit;
begin
    oTEvent1.vSignal;              {signal timer event to exit the thread}
    {wait until the thread exits}
    while (GetThreadState(hTHandle1) > nentTSTerm) do
        Wait(25);
    endwhile;

    {delete the thread}
    if not DestroyThread(hTHandle1) then
        Writeln("Failed to delete thread 1");
    endif
    oTEvent1.vDone;
end;

{initalisation}
begin
    oTEvent1.poInit(1000);          {timer event every 1000ms}
end.

```

9 Monitor and keyboard functions

9.1 Introduction

This chapter describes the monitor and keyboard functions. There are functions to affect the input/output of the standard output of the AIDA Commanders and the processing of keyboard events

You can create a command line user interface using the input/output functions. Additionally there are commands to control the Commander by other programs.

During keyboard event processing you can retrieve the key code of the hitted key and perform the required tasks. There are two types of keyboard events. Wait events block program execution until the event occurs and NoWait events do not block. Instead a call to the WaitEvent(s) function handles an occurred event (see the chapter on events).

9.2 Descriptions

Declaration

```
procedure ClrEol;
```

Parameter

None.

Return value

None.

Description

Deletes the data from the current cursor position to the end of the line.

Remarks

None

Example

```
procedure vMain;  
begin  
  ClrEol;           {delete data until end of line}  
end;
```

ClrScr**Declaration**

```
procedure ClrScr;
```

Parameter

None.

Return value

None.

Description

Clears the screen.

Remarks

None.

Example

```
procedure vMain;
var
  wKey:    Word;
  csTest:  String;
  i32Test: Int32;

begin
  Writeln("Text to be deleted - 1.line");
  Writeln("Text to be deleted - 2.line");
  wKey := ReadKey; //wait for input (block program execution)
  ClrScr;          //clear screen
  wKey := ReadKey; //wait for input (block program execution)
end;
```

GetMaxX**Declaration**

```
function GetMaxX: Word;
```

Parameter

None.

Return value

Word.

Returns the width of the standart output (console).

Description

The function returns the width of the standard output. You can set the width of AIDA Commanders standard output using the menu Setup → Window size.

Remarks

None.

Example

```
procedure vMain;
begin
  Writeln(GetMaxX); // print width of the standard output
end;
```

GetMaxY**Declaration**

```
function GetMaxY: Word;
```

Parameter

None.

Return value

Word.

Returns the height of the standart output (console).

Description

The function returns the height (number of lines) of the standard output. You can set the height of AIDA Commanders standard output using the menu Setup → Window size.

Remarks

None.

Example

```
procedure vMain;
begin
  Writeln(GetMaxY); // print height of the standard output (# of lines)
end;
```

GetX**Declaration**

```
function GetX: Word;
```

Parameter

None.

Return value

Word.

Current X position of the cursor.

Description

The function returns the current X position of the cursor.

Remarks

None.

Example

```
procedure vMain;
begin
  Write("Text");
  Writeln(GetX); // prints the X position of the cursor. 5 in this case
end;
```

GetY**Declaration**

```
function GetY: Word;
```

Parameter

None.

Return value

Word.

Current Y position of the cursor.

Description

The function returns the current Y position of the cursor.

Remarks

None.

Example

```
procedure vMain;
begin
  Writeln("Text");
  Writeln(GetY); // prints the Y position of the cursor. 2 in this case.
end;
```

GotoXY**Declaration**

```
procedure GotoXY (x,y: Word);
```

Parameter

x.

Word. Desired X position of the cursor.

y.

Word. Desired Y position (line) of the cursor.

Return value

None.

Description

The procedure positions the cursor to the specified X/Y position.

Remarks

None.

Example

```
procedure vMain;
var
  wKey: Word;

begin
  GotoXY(15,10); {set the cursor to position 15 in line 10}
  wKey := ReadKey; {wait for keyboard input (block program execution)}
end;
```

Newln**Declaration**

```
procedure Newln;
```

Parameter

None.

Return value

None.

Description

The procedure sets the cursor to the beginning of the next line.

Remarks

None.

Example

```
procedure vMain;
var
  wKey: Word;
begin
  wKey := ReadKey; {wait for keyboard input (block program execution)}
  Newln;          {set the cursor to the beginning of the next line}
  wKey := ReadKey; {wait for keyboard input (block program execution)}
end;
```

IDECommand

Declaration

```
function IDECommand (csCmd: String; ..): String;
```

Parameter

csCmd.

String. Contains the command that is passed to the terminal emulator.

..

untyped list of parameters.

The parameters are converted into a comma separated string when creating a message that is passed to the AIDA Commander.

Return value

String.

Resulting string of the AIDA Commander.

Description

The function creates an internal command to the terminal emulator (AIDA Commander). The command is passed via the string csCmd. The list of untyped command parameters is converted into a comma separated string.

These strings are passes to the AIDA Commander and a string is returned containing the result.

A description of the commands, their parameters and the possible return values is provided in the AIDA Commander reference. You can find the reference using the AIDA Commander help.

Remarks

None.

Example

```
procedure vMain;
var
  wKey:   Word;
  csTest: String;

begin
  csTest := "config.setMainBounds";           {change window size and position}
  Writeln(IDECommand(csTest,10,10,900,700)); {execute command and provide the
                                             parameters}
end;
```

IDECommand_NW

Declaration

```
procedure IDECommand_NW (var oNWEvent: toNWEvent; var csRsp: String;  
csCmd: String; ..);
```

Parameter

oNWEvent.

`toNWEvent` as reference. Event class of the NoWait object. The object has to be initialized using the constructor `polnit` before it is used and it has to be deleted using `vDone` before the program terminates.

The object contains the error code (`oNWEvent.dwErrNo`) after the event is signaled and possibly a result (`oNWEvent.unVal.dw`, `oNWEvent.unVal.i32`, or `oNWEvent.unVal.pv`). For details see the description of the `toEvent` object.

csRsp.

`String`. Result string of the AIDA Commander.

csCmd.

`String`. Contains the command that is passed to the terminal emulator.

..

Untyped list of parameters.

The parameters are converted into a comma separated string when creating a message that is passed to the AIDA Commander.

Return value

None.

Description

The function creates an internal command to the terminal emulator (AIDA Commander). The command is passed via the string `csCmd`. The list of untyped command parameters is converted into a comma separated string.

These strings are passes to the AIDA Commander and a result string is returned via the parameter `csRsp`.

A description of the commands, their parameters and the possible return values is provided in the AIDA Commander reference. You can find the reference using the AIDA Commander help.

IDECommand_NW

Remarks

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, resp. the appropriate NoWait versions).

The procedure IDECommand_NW does not wait for a return value contrary to the function IDECommand.

Example

(calling IDECommand using timeout)

```
module Test;

private
var
  oNWEvent: toNWEvent;
  oTEvent:  toTEvent;
  poEvent:  tpoEvent;

procedure vMain;
var
  csRes:      String;
  csCmdPar:   String;

begin
  csCmdPar := "config.setMainBounds"; {command for the AIDA Commander}
  {Aufruf des Befehls}
  IDECommand_NW(oNWEvent,csRes,csCmdPar,10,10,900,700);

  {as soon as the commad is executed or a timeout occurred}
  if WaitEvents(oNWEvent,oTEvent) = tpoEvent(@oNWEvent) then
    Writeln(csRes);           {print the result string}
  else
    Writeln("error calling AIDA command");
  endif;

end;

procedure vDeninit;
begin
  oNWEvent.vDone;
  oTEvent.vDone;
end;

begin
  oNWEvent.poInit;
  oTEvent.poInit(5000);
end.
```

KeyPressed

Declaration

```
function KeyPressed: Word;
```

Parameter

None.

Return value

Word.

Key code of the pressed key or 0 if no key was pressed. A description of the key codes is provided in chapter 9.3. werden.

Description

The function verifies whether a key was pressed and returns the key code of the pressed key. The key code remains in the internal buffer and the function returns immediately. If you want to wait for input use the function WaitKey resp. WaitKey_NW.

Remarks

Additional the variable unLastKeyVal is set to the pressed key.

Example

```
procedure vMain;
var
  wKey:      Word;
  i32Count: Int32;

begin
  for i32Count := 1 to 1000 do
    Writeln(i32Count); {print the loop counter}
    wKey := KeyPressed;

    if wKey <> 0 then {in case a key was pressed}
      Writeln(wKey); {prints the key code}
      break;        {exit the for loop}
    endif;
    Wait(200);
  endfor;
end;
```

WaitKey**Declaration**

```
function WaitKey (wTO: Word): Word;
```

Parameter

wTo.

Word. Timeout in ms or nwTimeoutInfinite (no timeout).

Return value

Word.

Key code of the pressed key or 0 if no key was pressed. A description of the key codes is provided in chapter 9.3.

Description

The function waits until a key is pressed and returns the key code of the pressed key. The key code remains in the internal buffer. If no key was pressed within wTo milli seconds the program continues despite no key was pressed. A wTo of nwTimeoutInfinite sets the time out to infinite.

Remarks

Additional the variable unLastKeyVal is set to the pressed key.

Example

```
procedure vMain;
begin
  Writeln(WaitKey(10000));           {wait on a key hit for 10 seconds}
  Writeln(WaitKey(nwTimeoutInfinite)); {wait inifite on a key hit}
end;
```

WaitKey_NW

Declaration

```
procedure WaitKey_NW (var oNWEvent: toNWEvent; wTO: Word);
```

Parameter

oNWEvent.

toNWEvent as reference (see also toNWEvent and toEvent). The object contains the error code (oNWEvent.dwErrNo) after the event is signaled and the result (oNWEvent.unVal.dw). For details see the description of the toEvent object.

wTo.

Word. Timeout in ms or nwTimeoutInfinite (no timeout).

Return value

None.

Description

The procedure uses a NoWait event to detect whether a key was pressed. The key code of the pressed key can be queried using oNWEvent.unVal.dw after the event is signaled. The key code remains in the internal buffer. If no key was pressed within wTo milli seconds the program continues despite no key was pressed. oNWEvent.unVal.dw is 0 in this case. A wTo of nwTimeoutInfinite sets the time out to infinite. To wait for the events use the WaitEvent(s) function. You can also signal the event using the vSignal procedure.

Remarks

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, resp. the appropriate NoWait versions).

Example

(see also the example of IDECommand_NW)

```
procedure vMain;
begin
  WaitKey_NW(oNWEvent,10000);           {wait for a key hit with timeout of
                                        10s}
  if (WaitEvent(oNWEvent)= nil) then   {wait for the event}
    Writeln(GetErrorMsg(GetError));    {in case of an error}
```

WaitKey_NW

```
    return;
  endif;
  Writeln("Keycode: ",oNWEvent.unVal.dw); {print the key code}
end;
```

ReadKey

Declaration

```
function ReadKey: Word;
```

Parameter

None.

Return value

Word.

Key code of the pressed key or 0 if no key was pressed.

Description

The function waits until a key is pressed and returns the key code of the pressed key. The ReadKey function has no timeout contrary to the function WaitKey.

Remarks

Additional the key code of the pressed key is stored in the variable unLastKeyVal.

Example

```
procedure vMain;
var
  wKey: Word;

begin
  wKey := ReadKey; {wait for a key being pressed}
  Writeln(wKey);   {print the key code}
end;
```

ReadKey_NW

Declaration

```
procedure ReadKey_NW(var oNWEvent: toNWEvent);
```

Parameter

oNWEvent.

toNWEvent as reference (see also toNWEvent and toEvent). The object contains the error code (oNWEvent.dwErrNo) and the key code (oNWEvent.unVal.dw) after the event was signaled.

Return value

None.

Description

The procedure detects keyboard hits using a NoWait event. The key code of the pressed key is stored in oNWEvent.unVal.dw after the event is signaled. To wait for the events use the WaitEvent(s) function. You can also signal the event using the vSignal procedure.

Remarks

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, resp. the appropriate NoWait versions).

Example

(see also the example of IDECommand_NW)

```
procedure vMain;
begin
  ReadKey_NW(oNWEvent);           {wait for keyboard input without
                                  timeout}
  if(WaitEvent(oNWEvent) = nil) then {waiting for the event}
    Writeln(GetErrorMsg(GetError)); {print error message}
    return;
  endif;
  Writeln("Keycode: ",oNWEvent.unVal.dw); {print key code}
end;
```

ReadKey_NW**Declaration**

```
function ReadStr(csDef: String): String;
```

Parameter

csDef.

String. csDef is put to the standard output. The passed string is marked and can be overwritten.

Return value

String.

Input read from standard input of empty string if no input was provided by the user.

Description

The function passes a string to the standard output. The passed string is marked and can be overwritten. The user input is returned when the return key is pressed or an empty string if no input was made by the user.

Remarks

None.

Example

```
procedure vMain;
var
  csTest: String;
  csRes: String;

begin
  csTest := "Enter text";
  csRes := ReadStr(csTest); {print: Enter text}
                                {after pressing return:}
  Writeln(csRes);           {print: entered text}
end;
```

GetLine**Declaration**

```
function GetLine (var cs: String; xoCursorOffs, xoMarkSize: tSize; bStackID:
Byte): Word;
```

Parameter

cs.

String as reference.

Input: A default string.

Output: A Result string. The default string is returned if no string is available.

xoCursorOffs.

tSize equals Int32. Start position of csDef's cursor. A 0 points to the beginning of the string.

bMarkSize.

tSize equals Int32. Number of characters selected. Use the constant MAX_SIZE to select the whole string.

bStackID.

Byte. ID of the command stack to use.

Possible values:

<pre>nGLSIDNone = 0; {no stack} nGLSIDCmdLine = 1; {POOL command line } nGLSIDReadStr = 2; {ReadStr function } {>2 für Applikation}</pre>
--

Return value

The key code that terminated the input. The scancode is always 0 for ASCII characters.

Description

The function reads a string from an editor. The parameter cs is a default string that is passed as reference. The parameters xoCursorOffs and xoMarkSize are used to mark a substring from xoCursorOffs top xoCursorOffs + xoMarkSize that can be overwritten. The input string is written to the variable cs after enter was pressed. The stack resp. command line to use is specified via bStackID. The function result is the key code that terminated the input.

GetLine**Remarks**

Additional the key code of the pressed key is stored in the variable unLastKeyVal.

Example

```
procedure vMain;
var
  csPar:          String;
  wRes:          Word;

begin
  csPar := "default string";
  wRes := GetLine (csPar,0,SIZE_MAX,1); {calling th function}
  Writeln("\n",csPar);                 {print the modified string}
  Writeln(wRes);                       {print the termination key code}
end;
```

GetLine_NW**Declaration**

```
procedure GetLine_NW (var oNWEvent: toNWEvent; var cs: String;  
                    xoCursorOffs, xoMarkSize: tSize; bStackID: Byte);
```

Parameter

oNWEvent.

`toNWEvent` as reference (see also `toNWEvent` and `toEvent`). The object contains the error code (`oNWEvent.dwErrNo`) and the function result (`oNWEvent.unVal.dw`) after the event was signaled. For details see the description of the object `toEvent`.

cs.

`String` as reference.

Input: A default string.

Output: A result string. The default string is returned if no string is available.

xoCursorOffs.

`tSize` equals `Int32`. Start position of `csDef`'s cursor. A 0 points to the beginning of the string.

bMarkSize.

`tSize` equals `Int32`. Number of characters selected. Use the constant `MAX_SIZE` to select the whole string.

bStackID.

`Byte`. ID of the command stack to use.

Possible values:

`nGLSIDNone` = 0: No stack.

`nGLSIDCmdLine` = 1: POOL command line.

`nGLSIDReadStr` = 2: `ReadStr` function.

>2: For applications.

Keyboard codes

Return value

The key code that terminated the input. The scancode is always 0 for ASCII characters.

Description

The function is used to read strings from the line editor. A default string `cs` is passed as reference. The parameters `xoCursorOffs` and `xoMarkSize` select a substring of `cs`. This selected substring can be overwritten. The result string is written to the variable `cs` after the return key was pressed.

The stack to use respective the command line to use is determined via the `bStackID` parameter. The function result is the key code that terminated the input.

Remarks

Additional the key code of the pressed key is stored in the variable `unLastKeyVal`.

Example

(see also the example of `IDECommand_NW`)

```
procedure vMain;
var
  csPar:          String;

begin
  csPar := "default string";
  GetLine_NW(oNWEvent, csPar, 0, SIZE_MAX, 1); {call the function}
  if (WaitEvent(oNWEvent) = nil) then        {wait for the event}
    Writeln(GetErrorMsg(GetError));          {print error message}
    return;
  endif;
  Writeln("\n", csPar);                       {print the modified string}
  Writeln(oNWEvent.unVal.dw);                 {print the termination key code}
end;
```

Keyboard codes

9.3 Keyboard codes

```

{Remarks: }
{Taken from the pool.pli library }
{tenScanCode: scan codes of the special keys }
{----- }
type tenScanCode = (
  nenSC000,      { 0 }
  nenSC001,      { 1, AltEsc? }
  nenSCAltSpace, { 2 }
  nenSCNul,      { 3, ^@ ? }
  nenSCBreak,    { 4, only temporary, to be determined! }
  nenSC005, nenSC006, nenSC007, nenSC008, nenSC009,
  nenSC010, nenSC011, nenSC012, nenSC013,
  nenSCAltBS,    { 14 }
  nenSCShTab,    { 15 }
  nenSCAltQ,     { 16 }
  nenSCAltW,     { 17 }
  nenSCAltE,     { 18 }
  nenSCAltR,     { 19 }
  nenSCAltT,     { 20 }
  nenSCAltY,     { 21 }
  nenSCAltU,     { 22 }
  nenSCAltI,     { 23 }
  nenSCAltO,     { 24 }
  nenSCAltP,     { 25 }
  nenSCAlt_91,   { 26 } { [ }
  nenSCAlt_93,   { 27 } { ] }
  nenSCAltCR,    { 28 }
  nenSC029,      { 29 }
  nenSCAltA,     { 30 }
  nenSCAltS,     { 31 }
  nenSCAltD,     { 32 }
  nenSCAltF,     { 33 }
  nenSCAltG,     { 34 }
  nenSCAltH,     { 35 }
  nenSCAltJ,     { 36 }
  nenSCAltK,     { 37 }
  nenSCAltL,     { 38 }
  nenSCAlt_59,   { 39 } { , }
  nenSCAlt_39,   { 40 } { ' }
  nenSCAlt_96,   { 41 } { ` }
  nenSC042,      { 42, not used }
  nenSCAlt_92,   { 43 } { \ }
  nenSCAltZ,     { 44 }
  nenSCAltX,     { 45 }
  nenSCAltC,     { 46 }
  nenSCAltV,     { 47 }
  nenSCAltB,     { 48 }
  nenSCAltN,     { 49 }
  nenSCAltM,     { 50 }
  nenSCAlt_44,   { 51 } { , }
  nenSCAlt_46,   { 52 } { . }
  nenSCAlt_47,   { 53 } { / }
  nenSC054,      { 54, not used }

```

nenSCAltN42,	{ 55 }	{ * NP }
nenSC056, nenSC057, nenSC058,		
nenSCF1,	{ 59 }	
nenSCF2,	{ 60 }	
nenSCF3,	{ 61 }	
nenSCF4,	{ 62 }	
nenSCF5,	{ 63 }	
nenSCF6,	{ 64 }	
nenSCF7,	{ 65 }	
nenSCF8,	{ 66 }	
nenSCF9,	{ 67 }	
nenSCF10,	{ 68 }	
nenSC069, nenSC070,		
nenSCHome,	{ 71 }	
nenSCCuUp,	{ 72 }	
nenSCPgUp,	{ 73 }	
nenSCAltN45,	{ 74 }	{ - NP }
nenSCCuLt,	{ 75 }	
nenSCCentCu,	{ 76 }	{ 5 NP }
nenSCCuRt,	{ 77 }	
nenSCAltN43,	{ 78 }	{ + NP }
nenSCEnd,	{ 79 }	
nenSCCuDn,	{ 80 }	
nenSCPgDn,	{ 81 }	
nenSCIns,	{ 82 }	
nenSCDel,	{ 83 }	
nenSCShF1,	{ 84 }	
nenSCShF2,	{ 85 }	
nenSCShF3,	{ 86 }	
nenSCShF4,	{ 87 }	
nenSCShF5,	{ 88 }	
nenSCShF6,	{ 89 }	
nenSCShF7,	{ 90 }	
nenSCShF8,	{ 91 }	
nenSCShF9,	{ 92 }	
nenSCShF10,	{ 93 }	
nenSCCtrlF1,	{ 94 }	
nenSCCtrlF2,	{ 95 }	
nenSCCtrlF3,	{ 96 }	
nenSCCtrlF4,	{ 97 }	
nenSCCtrlF5,	{ 98 }	
nenSCCtrlF6,	{ 99 }	
nenSCCtrlF7,	{ 100 }	
nenSCCtrlF8,	{ 101 }	
nenSCCtrlF9,	{ 102 }	
nenSCCtrlF10,	{ 103 }	
nenSCAltF1,	{ 104 }	
nenSCAltF2,	{ 105 }	
nenSCAltF3,	{ 106 }	
nenSCAltF4,	{ 107 }	
nenSCAltF5,	{ 108 }	
nenSCAltF6,	{ 109 }	
nenSCAltF7,	{ 110 }	
nenSCAltF8,	{ 111 }	
nenSCAltF9,	{ 112 }	
nenSCAltF10,	{ 113 }	

```

nenSCCtrlPrtSc, { 114 }
nenSCCtrlCuLt, { 115, often used as "word left" }
nenSCCtrlCuRt, { 116, often used as "word right" }
nenSCCtrlEnd, { 117 }
nenSCCtrlPgDn, { 118 }
nenSCCtrlHome, { 119 }
nenSCAlt1, { 120 }
nenSCAlt2, { 121 }
nenSCAlt3, { 122 }
nenSCAlt4, { 123 }
nenSCAlt5, { 124 }
nenSCAlt6, { 125 }
nenSCAlt7, { 126 }
nenSCAlt8, { 127 }
nenSCAlt9, { 128 }
nenSCAlt0, { 129 }
nenSCAlt_45, { 130 } { - }
nenSCAlt_61, { 131 } { = }
nenSCCtrlPgUp, { 132 }
nenSCF11, { 133 }
nenSCF12, { 134 }
nenSCShF11, { 135 }
nenSCShF12, { 136 }
nenSCCtrlF11, { 137 }
nenSCCtrlF12, { 138 }
nenSCAltF11, { 139 }
nenSCAltF12, { 140 }
nenSCCtrlCuUp, { 141 }
nenSCCtrlN45, { 142 } { - NP }
nenSCCtrlCCu, { 143 } { 5 NP }
nenSCCtrlN43, { 144 } { + NP }
nenSCCtrlCuDn, { 145 }
nenSCCtrlIns, { 146 }
nenSCCtrlDel, { 147 }
nenSCCtrlTab, { 148 }
nenSCCtrlN47, { 149 } { / NP }
nenSCCtrlN42, { 150 } { * NP }
nenSCAltHome, { 151 }
nenSCAltCuUp, { 152 }
nenSCAltPgUp, { 153 }
nenSC154, { 154, not used }
nenSCAltCuLt, { 155 }
nenSC156, { 156, not used }
nenSCAltCuRt, { 157 }
nenSC158, { 158, not used }
nenSCAltEnd, { 159 }
nenSCAltCuDn, { 160 }
nenSCAltPgDn, { 161 }
nenSCAltIns, { 162 }
nenSCAltDel, { 163 }
nenSCAltN47, { 164 } { / NP }
nenSCAltTab, { 165 }
nenSCAltEnter { 166 }
{ NP: Numeric Pad } );

```

10 Menu functions

Introduction

You can use menu functions to implement a user interface independent of the underlying operating system

MenuShow

Declaration

```
function MenuShow (csMenuItems: String; wPosX, wPosY: Word; i16DefLine:
Int16):Boolean;
```

Parameter

csMenuItems.

String. A list of menu elements separated using the character '|'. The first entry is the title of the menu. A hyphen creates a separator in the menu.

The example string: 'Calculator|Addition|Subtraction|-|Quit' provides following menu:



Figure 1: menu

wPosX.

Word. X position of the left top corner of the menu in units of characters of the underlying text window.

wPosY.

Word. Y position of the left top corner of the menu in units of characters of the underlying text window.

i16DefLine.

Int16. Line to highlight as default (starting at 1). No menu entry is highlighted in case of an invalid value.

Return value

Boolean.

`false` in case of an empty string or if an error occurs, otherwise `true`.

Description

The function shows the menu as the position wPosX, wPosY. The menu entries are passed using a string of entries separated by the '|' character. You can create separators using a hyphen. The parameter i16DefLine indicates which menu entry is to

MenuShow

highlight as default menu selection. The function returns `true` if successful and `false` otherwise.

Remarks

None.

Example

```
procedure vMain;

var
  i16MenuChoice: Int16;
  r64Value1:    Real64;
  r64Value2:    Real64;

begin
  Writeln("CALCULATOR: ");
  repeat
    {show the menu}
    if !MenuShow('Calculator|Addition|Subtraction|-|Quit',50,10,1) then
      Writeln(GetErrorMsg(GetError));
      return;
    endif;

    {retrieve user selection (after return key was pressed)}
    i16MenuChoice := MenuExecute;

    {close the menu after entry was selected}
    if !MenuClose then
      Writeln(GetErrorMsg(GetError));
      return;
    endif;

    {handle selected menu entry}
    case i16MenuChoice of
      1: begin
          {addition selected}
          {read the first number:}
          Writeln("\nEnter a number: ");
          r64Value1 := R64Val(ReadStr(''));

          {read the second number:}
          Writeln("\nEnter a second number: ");
          r64Value2 := R64Val(ReadStr(''));

          {print addition's result}
          Writeln("\n\nAddition result: ",r64Value1 + r64Value2);
        end;
      2: begin
          {subtraction selected}
          {read the first number:}
          Writeln("\nEnter a number: ");
          r64Value1 := R64Val(ReadStr(''));

          {read a second number:}

```

MenuShow

```
        Writeln("\nEnter a second number: ");
        r64Value2 := R64Val(ReadStr(''));

        {print subtraction's result}
        Writeln("\n\nSubtraction result: ",r64Value1 - r64Value2);
    end;

    3: begin                                {quit selected}
        break
    end;

    else                                    {no valid selection}
        Writeln("Calculator|Addition|Subtraction|-|Quit ");
    endcase;

    until false;
end;
```

MenuExecute

Declaration

```
function MenuExecute: Int16;
```

Parameter

None.

Return value

Int16.

Selected menu entry (1 refers to the line at the top).

Description

The function returns the menu entry the user selected as an `Int16` variable. A menu entry is selected using the up/down keys to select the entry and pressing enter or using the mouse alternatively. The entries are numbered sequentially starting at 1 for the first entry.

Remarks

None.

Example

(see also the function `MenuShow`)

```
{retrieve user selection (after return key was pressed)}
i16MenuChoice := MenuExecute;

{close the menu after entry was selected}
if !MenuClose then
  Writeln(GetErrorMsg(GetError));
  return;
endif;

{handle selected menu entry}
case i16MenuChoice of
  1: begin                                {1. line selected}
      {Anweisungen}
    end;
  2: begin                                {2. line selected}
      {Anweisungen}
    end;
  3: begin                                {3. line selected; terminate program}
      break
    end;
else                                     {no valid selection}
```

MenuExecute

```
        Writeln("Invalid menu selection");  
    endcase;
```

MenuExecute_NW

Declaration

```
procedure MenuExecute_NW(var oNWEvent: toNWEvent);
```

Parameter

oNWEvent.

toNWEvent as reference (see also toNWEvent and toEvent). The object contains the error code (oNWEvent.dwErrNo) and the function result (oNWEvent.unVal.i32) after the event was signaled. For details see the description of the object toEvent.

Return value

None.

Description

The procedure returns the selected menu via oNWEvent.unval.i32. A menu entry is selected using the up/down keys to select the entry and pressing enter or using the mouse alternatively. The entries are numbered sequentially starting at 1 for the first entry. MenuExecute_NW does not wait for the event contrary to MenuExecute. Instead the functions WaitEvent(s) can be used to wait for the event. You can also signal the event manually using vSignal.

Remarks

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, resp. the appropriate NoWait versions).

MenuExecute_NW does not wait for the event contrary to MenuExecute. Instead the functions WaitEvent(s) can be used to wait for the event

Example

(see also the function MenuShow)

```
module Test;

private
var
  oTEvent: toTEvent;
  oNWEvent: toNWEvent;

procedure vMain;
var
  i16MenuChoice: Int16;
```

MenuExecute_NW

```

r64Value1:   Real64;
r64Value2:   Real64;

begin
  Writeln("CALCULATOR: ");
  repeat
    {show the menu}
    if !MenuShow('Calculator|Addition|Subtraction|-|Quit',50,10,1) then
      Writeln(GetErrorMsg(GetError));
      return;
    endif;

    {wait for user input using a timeout}
    MenuExecute_NW(oNWEvent);

    if (WaitEvents(oNWEvent,oTEvent))= nil then
      Writeln("Error occurred -",GetErrorMsg(GetError));
      break;
    endif;

    case GetEventNo of
    1: begin
      {menu event}
      i16MenuChoice := Int16(oNWEvent.unVal.i32);
      if !MenuClose then
        Writeln(GetErrorMsg(GetError));
        return;
      endif;

      case i16MenuChoice of
      1: begin
        {addition selected}
        {read the first number:}
        Writeln("\nEnter a number: ");
        r64Value1 := R64Val(ReadStr(''));

        {read the second number:}
        Writeln("\nEnter a second number: ");
        r64Value2 := R64Val(ReadStr(''));

        {print addition's result}
        Writeln("\n\nAddition result: ", r64Value1 + r64Value2);
        oTEvent.vDone;           {Destruktor}
        oTEvent.poInit(5000);    {restart of the time}

      end;

      2: begin
        {subtraction selected}
        {read the first number:}
        Writeln("\nEnter a number: ");
        r64Value1 := R64Val(ReadStr(''));

        {read the second number:}
        Writeln("\nEnter a second number: ");
        r64Value2 := R64Val(ReadStr(''));

        {print subtractions's result}
        Writeln("\n\nSubtraction result: ", r64Value1 - r64Value2);
        oTEvent.vDone;           {destructor}
        oTEvent.poInit(5000);    {restart timer}
      end;
    end;
  end;
end;

```

MenuExecute_NW

```
        end;

        3: begin                                {quit program selected}
            break
        end;

        else                                    {no valid menu selected}
            Writeln("No valid menu selected");

        endcase;                                {end menu selection}
    end;

    2: begin                                    {timer event}
        break;                                  {exit program}
    end;

    else
        begin
            break;
        end;
    endcase;                                    {end of the event handling}

until false;

end;

procedure vDeinit;
begin
    oNWEvent.vDone;
    oTEvent.vDone;
end;

begin
    oTEvent.poInit(5000);
    oNWEvent.poInit;
end.
```

MenuClose**Declaration**

```
function MenuClose: Boolean; ifr;
```

Parameter

None.

Return value

Boolean.

`true` if successful, otherwise `false`.

Description

The function closes a menu that was opened using `ShowMenu`. `true` is returned if the menu was closed successfully, otherwise `false`.

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

(see also function `MenuShow`)

```
MenuShow('Calculator|Addition|Subtraction|-|Quit',50,10,5);

{retrieve user selection (after return key was pressed)}
i16MenuChoice := MenuExecute;

{close menu - including error handling}
if !MenuClose then
  Writeln(GetErrorMsg(GetError));
  return;
endif;
```

MenuCloseAll**Declaration**

```
function MenuCloseAll: Boolean; ifr;
```

Parameter

None.

Return value

Boolean.

`true`, if all menus were closed successfully, otherwise `false`.

Description

The function closes all menus. `True` is returned if successful, otherwise `false`.

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

(see also function `MenuShow`)

```
MenuShow('Calculator|Addition|Subtraction|-|Quit',50,10,1);
{retrieve user selection (after return key was pressed)}
i16MenuChoice1 := MenuExecute;

MenuShow('Data typ|Int32|Real64|',60,10,1);
{retrieve user selection (after return key was pressed)}
i16MenuChoice2 := MenuExecute;

{close all menus}
if !MenuCloseAll then
  Writeln(GetErrorMsg(GetError));
  return;
endif;
```

MenuCount**Declaration**

```
function MenuCount: Int16;
```

Parameter

None.

Return value

Int16.

Number of open menus.

Description

The function returns the number of open menus as Int16 value.

Remarks

None.

Example

(see also functionMenuShow)

```
{show menu}
MenuShow('Calculator|Addition|Subtraction|-|Quit',50,10,1);
Writeln(MenuCount);           {prints: 1}

{retrieve user selection (after return key was pressed)}
i16MenuChoice1 := MenuExecute;
MenuShow('Data type|Int32|Real64|',60,10,1);
Writeln(MenuCount);           {prints: 2}

{retrieve user selection (after return key was pressed)}
i16MenuChoice2 := MenuExecute;

{close all menus}
MenuCloseAll;
Writeln(MenuCount);           {prints: 0}
```

11 Thread and task functions

Taken from the pool.pli library.

Introduction

Please see the appropriate subsections for a brief description of the basics on how to use thread and task functions.

11.1 Threads

Introduction

Since working with threads is a very complex topic, the introduction can only provide a short overview. If you would like to or have to know more about threads, please turn to additional literature such as [1].

Terms

Process

Instance of a started program, to which all resources of the application are allocated. A process can have several threads. There is always at least one thread for the execution of a process. It is called primary thread and represents the execution of the process.

Thread

The idea behind threads is to split a program (or process) into different program parts (program threads), which run independently from one another. Multitasking systems support this concept by making it possible for each process to start threads. A time slice of the processor is then allocated to these threads.

Programs that accept user data in the foreground and process them in the background are typical applications for threads. The application can react to user entries due to the division into two threads, while data is processed "simultaneously."

Other common applications are programs that work as a server and have to serve several clients at the same time. A separate thread is started for each client that connects to the server, and this thread will serve the client regardless of the remainder of the process.

Operating Principle of Threads

Modern operating system such as Windows and UNIX work with the time slicing method according to the multitasking principles. This means that several processes resp. threads share one processor and run more or less in parallel. In this method each thread is assigned a time slice, within which it can freely work with the processor. In other words, first Thread1 is executed for 20ms for instance, then Thread2, then Thread3, and so on until all threads are processed. Only when all threads (of all processes) have received their share of processor time, will it be Thread1's turn again (we will omit the significance of different priorities at this point). The user will be under the impression that all programs or processes run simultaneously.

The operating system switches between the individual threads. Since the switch occurs without the help of the application, we are talking about "preemptive" multitasking.

Thus, the programmer can split his/her program into several threads without having to deal with the allocation of time slices. However, splitting only makes sense if programs have to run in parallel (see also the explanation of the term thread in chapter 11).

As discussed earlier, each process consists of at least one thread, the primary thread of the application. Any number of additional threads can be started from this primary thread. All threads of a process share its resources such as memory, files, communication channels, etc. Access to the resources (e.g., global variables) of other threads of the same process is possible anytime; however, this is not true for the access to resources of threads from other processes!

This is due to the fact that individual processes and thus also their threads are strictly separated from one another by the operating system and work in different virtual address areas. See also chapter 21, in which the shared memory functions of the shm.pli library are described.

The common access to the same resources of a process from within various threads can lead to problems. For this reason it is often necessary to synchronize the threads. Chapter 12 describes how this is done.

Each thread within a process has a separate stack, on which it stores its local variables. Only the thread can access these local variables. This is important in particular with client server architectures, where each client needs its own local variables.

Since the necessary reaction time of individual threads can vary, multitasking operating systems have a priority system, in which individual priority stages can be allocated. These priorities have an influence on the allocation of processing time. In more simple terms, high-priority threads are preferred over lower-priority threads during the processor allocation. Since it is currently only possible to select one priority stage in pool.pli (normal priority), we will not go into further details. You can look up additional information on this topic in [1].

Working with threads

In order to start a thread, the `CreateThread` function has to be called using the appropriate priority (currently only 0 = normal priority). It returns a handle to the new thread, which identifies the thread. You should use an if statement to check, whether the thread was successfully created and whether you are now working with the new thread. Usually a function is called within the thread, in which the actual thread processing is implemented. It is important at this point to understand, that you have two "program parts" that are running independently from one another.

There are functions for working with threads that can be used to retrieve the handle of the current thread, the thread status, and the thread error code. More on this topic in the description of the individual functions.

A thread is completed after its commands were processed. Threads that have to run in "endless loops" are usually terminated by signaling an event. This event will cause the termination of the loop. The `DestroyThread` function has to be called after the thread is terminated, with the thread's handle as parameter. Thus internal administration structures of the thread are deleted by calling this function, and memory is deallocated.

An active thread can be terminated using the `TerminateThread` function. However, this function should not be used to ensure that the thread is terminated properly.

A detailed description of the functions is provided in the following section:

CreateThread

Declaration

```
function CreateThread(bFlags: Byte): tTHandle;
```

Parameter

bFlags.

Byte. Various flags, e.g. priority (currently only 0 = normal priority).

Return value

tTHandle.

Handle of the new thread or nInvHandle in case of an error.

Description

The function creates and starts a thread. You can pass the desired priority of the thread as parameter (currently only 0 = normal priority is supported). The function returns a handle of the new thread or nInvHandle in case of an error.

Remarks

It is not allowed to pass CharString or ByteString parameters by value to the actual function. Local data of type CharString or ByteString has to be empty (except static) in the point of time when CreateThread is called. This is valid for any local data of any object type (especially mutex and event objects), i.e. the objects have to be uninitialized.

The thread starts code execution directly after the call to CreateThread.

By comparing the result of CreateThread with the current thread (GetThreadHandle) you can distinguish between the old thread (<>) and the new thread (=).

Notice that the evaluation order is not determined. Therefore you should store the CreateThread result in a variable and then use this variable for comparison.

Example

```
xTHandle := CreateThread(0);  
if xTHandle = GetThreadHandle then..
```

Access to parameter or local variables (except static) of superior functions are not allowed. The thread ends at the end of the actual function or with Halt, in any case without a call of vDeinit. Even a terminated thread occupies memory to store its state and exit code. For this reason you must delete each thread using DestroyTread, especially when a thread was started multiple times.

CreateThread

Example

Remarks: vThread1 and vThread2 are procedures that are called within a thread and so they can be used multiple times (e.g. in client/server applications). Each call to CreateThread creates a separate instance and stack.

The following example creates an extra function for each spawned thread to support comprehension of the use of the functions. Both threads access a shared global variable

```

module Test;

procedure vCreateThread;
procedure vThread1;
procedure vThread2;

private
var
  hTHandle1:  tTHandle;           {handle of the first thread}
  hTHandle2:  tTHandle;           {handle of th second thread}
  oWriteMutex: toMutex;          {used for thread synchronisation}
  oTEvent1:   toTEvent;           {timer 1}
  oTEvent2:   toTEvent;           {timer 2}
  i32Value:   Int32;              {shared global variable}

{create two threads}
procedure vCreateThread;
begin
  hTHandle1 := CreateThread(0);   {create first thread}

  if(hTHandle1 = nInvHandle) then {in case of an error}
    Writeln("Error Thread 1-",GetErrorMsg(GetError));
  elseif (hTHandle1 = GetThreadHandle) then
    vThread1;                      {thread 1 call function}
    return;                          {end thread 1}
  endif

  hTHandle2 := CreateThread(0);   {create second thread}
  if(hTHandle2 = nInvHandle) then {in case of an error}
    Writeln("Error Thread 2-",GetErrorMsg(GetError));
  elseif (hTHandle2 = GetThreadHandle) then
    vThread2;                      {thread 2 call function}
    return;                          {end thread 2}
  endif
  return;
end;

{source code for the first thread}
procedure vThread1;
begin
  repeat
    oWriteMutex.vLock;              {synchronize access to i32Value}
    i32Value := i32Value + 1;
    Writeln("Thread1: ",i32Value);
  end;
end;

```

CreateThread

```

oWriteMutex.vUnlock;
if (WaitEvent(oTEvent1) = nil) then
  Writeln(GetErrorMsg(GetError));
  return;
endif;
until GetEventSignaled;
Writeln("Exit thread1");
end;

{source code for the second thread}
procedure vThread2;
begin
  repeat
    oWriteMutex.vLock;           {synchronize access to i32Value }
    i32Value := i32Value + 1;
    Writeln("Thread2: ", i32Value);
    oWriteMutex.vUnlock;
    if (WaitEvent(oTEvent2) = nil) then
      Writeln(GetErrorMsg(GetError));
      return;
    endif;
  until GetEventSignaled;
  Writeln("Exit thread2");
end;

procedure vMain;
begin
  vCreateThread;           {initialize threads }
  repeat                   {run threads}
    Wait(30);
  until KeyPressed <> 0;   {terminate program when key was pressed}
end;

{deinitialisation}
procedure vDeinit;
begin
  {signal timer events to terminate threads }
  oTEvent1.vSignal;
  oTEvent2.vSignal;

  {wait until first thread is terminated}
  while (GetThreadState(hTHandle1) > nenTSTerm) do
    Wait(25);
  endwhile;

  {delete first thread}
  if not DestroyThread(hTHandle1) then
    Writeln("Could not delete thread 1");
  endif

  {wait until second thread is terminated}
  while (GetThreadState(hTHandle2) > nenTSTerm) do
    Wait(25);
  endwhile;

```

CreateThread

```
{delete second thread}
if not DestroyThread(hTHandle2) then
  Writeln("Could not delete thread 2");
endif

{call destructors}
oTEvent1.vDone;
oTEvent2.vDone;
oWriteMutex.vDone;
end;

{initialisation}
begin
  oTEvent1.poInit(500);
  oTEvent2.poInit(500);
  oWriteMutex.poInit;
end.
```

DestroyThread

Declaration

```
function DestroyThread(var hTHandle: tTHandle): Boolean;
```

Parameter

hTHandle.

tTHandle as reference. Handle of the thread to be deleted.

Return value

Boolean.

true if thread was deleted successfully. false if the handle was invalid or the thread was in a state that did not allow to terminate it. For instance it is not possible to delete a thread that is not terminated.

Description

The function deletes a thread and its internal data structures. A thread has to be terminated before you can delete it.

Remarks

Even a terminated thread occupies memory to store its state and exit code. For this reason you must delete each thread using DestroyThread to delete this internal data and to free the memory.

Example

(see also CreateThread)

```
procedure vDeinit;
begin
  {delete the thread - the thread has to be terminated (see GetThreadState!)}
  if not DestroyThread(hTHandle1) then
    Writeln("Could not delete thread 1");
  endif
end;
```

GetThreadHandle

Declaration

```
function GetThreadHandle: tTHandle;
```

Parameter

None.

Return value

tTHandle.

Handle of the current thread. GetThreadHandle returns hOwnTHandle for the main thread of a process. The result is nInvHandle in case of an invalid handle (see example).

Description

The function returns the handle of the current thread. By comparing the result of CreateThread with the result of GetThreadHandle you can distinguish between the old thread (<>) and the new thread (=).

Notice that the evaluation order of the functions is not determined. Therefore you should store the result of CreateThread in a variable and use the variable to test whether you are in the new thread or not.

Remarks

None.

Example

(see also CreateThread)

```
procedure vCreateThread;
begin
  hTHandle := CreateThread(0);           {create a thread}
  if(hTHandle = nInvHandle) then        {in case of an error}
    Writeln("Error Thread 1-",GetErrorMsg(GetError));
  elseif (hTHandle = GetThreadHandle) then {within the new thread?}
    vThread;                             {call the threads function}
  return;                                 {terminate the thread}
  endif
end;
```

GetThreadState

Declaration

```
function GetThreadState(hTHandle: tTHandle): tenTState;
```

Parameter

tTHandle.

Handle des aktuellen Threads.

Return value

tenTState

nenTSNone: Thread does not exist.

nenTSTerm: Thread terminated, but state and exit code are valid.

nenTSLoad: Thread is loaded.

nTSWaitEvent: Thread is waiting for events.

nenTSWaitContinue: Thread is waiting to continue (breakpoint/single step).

nenTSActive: Thread is active and on the scheduler list.

Description

The function returns the state of the thread with the handle tTHandle.

Remarks

The state of the main tread is returned if the task handle is passed as parameter (see chapter 11.2).

Example

(see also CreateThread)

```
procedure vDeinit;
begin
    {signal timer to terminate threads}
    oTEvent.vSignal;

    {wait until thread is terminated}
    while (GetThreadState(hTHandle) > nenTSTerm) do
        Wait(25);
    endwhile;

    {delete the thread}
    if not DestroyThread(hTHandle) then
        Writeln("Could not delete thread");
```

GetThreadState

```
    endif  
end;
```

GetThreadExit-Code**Declaration**

```
function GetThreadExit-Code(hTHandle: tTHandle): Int32;
```

Parameter

tTHandle.

Handle of the thread.

Return value

Int32.

Exit code of the thread. -1 is returned in case of an error e.g. a non existing thread or Thread-Status <> nTSTerm.

Exit codes:

nExitSuccess	= 0:	Thread terminated without error.
nExitWarning	= 1:	Warning.
nExitAppError1	= 2:	Application error 1.
nExitAppError2	= 3:	Application error 2 (also Break).
nExitSysError	= 4:	System error (e.g. File-I/O).
nExitFatalError	= 5:	Internal error.

Description

The function returns the exit code of the thread.

Remarks

The state of the main tread is returned if the task handle is passed as parameter (see chapter 11.2). If the task has multiple threads, you have to evaluate the exit code of the threads in vDeinit yourself if necessary.

A call to the function with the thread's own handle is always possible.

Calling the function with the handle of an independent thread is not allowed. An independent thread can be started e.g. from the command line.

Example

(see also CreateThread and SetExitCode)

```
procedure vDeinit;  
begin  
  
    {signal a timer to terminate the thread}
```

GetThreadExit-Code

```
oTEvent.vSignal;  
  
{wait until the thread is terminated}  
while (GetThreadState(hTHandle) > nentTSTerm) do  
    Wait(25);  
endwhile;  
Writeln(GetThreadExitCode(hTHandle1)); {prints: 0 (since the is no error)}  
  
{delete the thread}  
if not DestroyThread(hTHandle) then  
    Writeln("Could not delete the thread");  
endif  
end;
```

TerminateThread

Declaration

```
function TerminateThread(hTHandle: tTHandle): Boolean;
```

Parameter

tTHandle.

Handle of the thread (task) to be terminated.

Return value

Boolean.

`true` if the termination was successful. `false` in case of an invalid handle or if the thread is in an state that does not allow to terminate the thread.

Description

The function terminates a thread or task.

Remarks

A thread can terminate itself using `TerminateThread`!

Threads and tasks should not be terminated using `TerminateThread`. Instead you should use events or command to terminate them.

Termination is analogous to the command `Halt` (seechapter 16.1).

The function has no effect if the termination of the main thread is already performing. In this case `true` is returned.

Example

```
procedure vMain;
begin
  vTestTask;           {starting a task from a subprogram -
                       the task prints a text ever 100ms
                       (no source code provided)}

  repeat
    Writeln("Hauptprogramm");
    Wait(25);
  until(KeyPressed <> 0); {until the program exits}

  Writeln("Terminate: ", TerminateThread(hSWHandle));
end;
```

11.2 Tasks

Introduction

You should read the introduction on thread in chapter 11 if you have not yet, for a better comprehension of the explanations provided in this section.

A task is an independent program (module) that is started and terminated from another program. A task is an independent program and therefore has at least a main thread whose handle is returned by the function `LoadTask`.

A task has an own main function and virtual address space contrary to a `Thread` (see also [1]). This is the reason why tasks can not access each others global variables. Data exchange requires shared memory (see chapter 21) and the according functions of the `shm.pli`.

You have to start a successfully loaded task using the `StartTask` function. With this call the task gets its independence from the program that loaded the task. The task is running until the task terminates itself, i.e. the task is not terminated by the calling program. (Even so it is possible to terminate the task from the calling program using `TerminateThread`, but as stated before, this function should not be used) To terminate the task from the calling program use a shared memory event (see chapter 21) to indicate to the task to terminate itself.

A task has to be terminated to delete it successfully. As for this the calling program should wait for the termination of the task within a loop (in general within the deinitialisation). Use the `getThreadState` function passing the task's main thread handle to determine whether the task is terminated or not. The task is terminated if the result is `nenTSNone` and with the function `UnloadTask` you can delete and free internal resources.

LoadTask

Declaration

```
function LoadTask(csName, csArg: String; xoStkLen: tSize; i8Mode: Int8):  
tTHandle;
```

Parameter

csName.

String. The name of the pi file (module) that is to be loaded as independent task.

csArg.

String. The task's argument string. The variable csArg is declared in pool.pli, and can be used in the task without additional declaration (see example).

xoStkLen.

tSize equals *Int32*. Desired POOL stack size. A default is used if xoStkLen <= 0 (recommended).

i8Mode.

Int8. Various modes, e.g. for console and priority. Currently only 0 is supported setting normal priority and a link to the current console.

Return value

tTHandle.

Handle of the new task. *nInvHandle* in case of an error.

Description

The function loads a task and passes the argument string to the task.

Remarks

The function merely loads the task without starting the task. To start the task use the function `StartTask`.

The AIDA runtime is terminated if a task causes a serious error during execution.

The returned task handle is also the thread handle of the task's main thread. The main thread is the only thread if no other thread is created within the task.

LoadTask**Example**

```
{load module TaskTest.pi as task}
hSWHandle := LoadTask("TaskTest","Teststring",0,0);
if hSWHandle = nil then           {in case of an error}
  Writeln("Could not load task - ",GetErrorMsg(GetError));
  return;
endif;
```

StartTask

Declaration

```
function StartTask(hTHandle: tTHandle): Boolean;
```

Parameter

tTHandle.

Handle of the task to start.

Return value

Boolean.

`true` if the task was started successfully. `false` in case of an invalid handle or if the task is in an invalid state to be started.

Description

The function starts a task. This task has to be loaded previously by a call to the `LoadTask` function. `true` is returned if the function was successful, `false` otherwise.

Remarks

Each task can be started only once after a `LoadTask` call.

Example

```
{start task}
if not StartTask(hSWHandle) then {in case the start fails}
  WritelnStr("Could not start task - ",GetErrorMsg(GetError));
endif;
```

The task that is to be started:

```
module TaskTest;

private
var
  i32Count: Int32;
procedure vMain;
begin
  Writeln(csArg);          {print argument string (csArg is defined in pool.pli)}
  for i32Count := 1 to 20 do
    Writeln("Task läuft");
    Wait(1000);
  endfor
  return;                  {program exits after ca. 20s - for another
```

StartTask

```
end;                                possibility see the chapter on shared memory}
```

UnloadTask

Declaration

```
function UnloadTask(var hTHandle: tTHandle): Boolean;
```

Parameter

hTHandle.

tTHandle as reference. Handle of the task that is to be unloaded.

Return value

Boolean.

true if the task was unloaded successfully. false in case of an invalid handle or if the task is in an invalid state to unload.

Description

The function gets a handle to a task. The task is unloaded and true is returned if the task was terminated. false is returned in case of an invalid handle or if the task is in an invalid state to unload.

Remarks

You can not unload a task until it is terminated and deinitialized. The task itself is responsible for its termination.

Internal structures used for administration purposes are freed when the task unloads.

The task handle is deleted after the task was unloaded successfully.

Example

```
procedure vDeinit;
begin
  while GetThreadState(hSWHandle) > nentTSTerm do {wait until task is
                                                    terminated}
    Wait(100);
  endwhile;

  if !(UnloadTask(hSWHandle)) then {unload the task}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
end;
```

GetThreadState

Declaration

```
function GetThreadState(hTHandle: tTHandle): tenTState;
```

Parameter

tTHandle.

hTHandle. Handle of the current task.

Return value

tenTState:

nenTSNone: Main thread does not exist.

nenTSTerm: Main thread is terminated and state and exit code are valid

nenTSLoad: Main thread is loaded.

nTSWaitEvent: Main thread is waiting for events.

nenTSWaitContinue: Main thread is waiting to continue (breakpoint/single step).

nenTSActive: Main thread is active and on the scheduler list.

Description

The function returns the state of the main thread.

Remarks

See also chapter 11 on threads.

Example

```
procedure vDeinit;
begin
  while GetThreadState(hSWHandle) > nenTSTerm do {wait until task is
                                                    terminated}
    Wait(100);
  endwhile;

  if !(UnloadTask(hSWHandle)) then {unload the task}
    Writeln(GetErrorMsg(GetError));
    return;
  endif;
end;
```

12 Synchronisation objects (Mutex)

Taken from the pool.pli library.

12.1 Introduction

If several processes or threads (see chapter11) share one resource, it becomes necessary to coordinate them via synchronization.

An example for this is two threads that access the same file, which was opened in the main program. The first thread positions the file pointer (see the appropriate section) to the data record that it wants to read. At that point in time the operating system switches to the second thread, which "redirects" the file pointer to another data record. Next, the first thread is used again and reads the data. However, it does not realize that the file pointer no longer points to the desired data and starts working with the wrong data.

The same problem occurs when using global variables with multiple threads. Let us assume thread1 wants to increment a global variable. To do this, it gets the global variable from the memory into its working register. Now it is thread2's turn, which also wants to increment the variable and loads it in its working register increments the value and writes it back to memory. Next, thread1 receives the processor again, increments the variable (without realizing that thread2 modified the variable), and increments the global variable overwriting the value that was stored by thread2. In other words, the variable was only increased by 1 instead of by 2.

Since errors like this one only occur sporadically and also depend on the operating rate of the processor, they are easily overlooked during program tests and lead to hard to find programming errors.

Thus, POOL offers so-called mutex objects, which are used to synchronize processes and threads. They are a kind of flag that is set by an object during the access to a resource, which is only reset once the resource is no longer needed. If another thread wants to access this resource, it checks whether the flag (the mutex) has been marked free. If it is free, it marks the flag occupied and accesses the resource. Otherwise, it either aborts the access or waits for it to be enabled. However, setting a flag does not prevent a thread, which does not check the flag from accessing the resource.

12.2 Mutex objects

Object toMutex

Definition

```
type
  toMutex = object (toEMRoot)
    constructor poInit;           {initialize private elements}
    procedure   vLock;           {lock mutex if necessary wait until mutex
                                is unlocked}
    function    boTryLock: Boolean; {try to lock the mutex:
                                    true=ok, false=not locked }
    procedure   vUnlock;         {unlock mutex}
  end; {toMutex}
  tpoMutex = ^toMutex;
```

Pointer to the object

tpoMutex.

Parent object

toEMRoot.

Description

Mutex objects are used to synchronize the access to shared resources with multiple threads (see also introduction).

A thread can use the functions `vLock` or `boTryLock` to reserve a mutex to access a shared resource. A reserved mutex can be released using `vUnlock`. If another thread tries to reserve a reserved mutex using `vLock` this thread waits until the reserved mutex is released by its current owner. If the thread uses `boTryLock` instead, the function returns immediately returning `false`. In this case it is possible that the thread never gets the mutex if other threads are waiting for the mutex using a `vLock` call. `vLock` has a `NoWait` counterpart, that uses the function `WaitEvent(s)` to wait for the mutex (see events in chapter 8).

You can prevent simultaneous access to a resource by acquiring access to the mutex object prior accessing the shared resource. Notice: `vLock` locks the mutex object not the resource itself.

toMutex**Remarks**

None

Methods

polnit.

Constructor used to initialize private elements to the parent class.

vLock.

Lock the mutex.

boTryLock.

Try to lock the mutex.

vUnlock.

Unlock the mutex.

toMutex**Description of the methods****Method declaration**

```
constructor poInit;
```

Parameter

None.

Return value

None.

Description

Constructor used to initialize private elements to the parent class.

Remarks

Each mutex object has to be initialized using a constructor call before it can be used.

Example

```
{initialisation}  
begin  
  oWriteMutex.poInit;  
end.
```

toMutex**Method declaration**

```
procedure vLock;
```

Parameter

None.

Return value

None.

Description

The procedure lock the mutex, or waits until the mutex is free.

Remarks

Each call of vLock needs a call of vUnlock within the same thread. A violation of this rule could lead to a deadlock situation where another thread is waiting for the mutex until the program terminates.

Example

```
{using a mutex in a thread}
oWriteMutex.vLock;    {lock the mutex to access a resource}
Writeln("Thread1");  {print to standard output}
oWriteMutex.vUnlock; {unlock the mutex}
```

toMutex**Method declaration**

```
procedure boTryLock;
```

Parameter

None.

Return value

Boolean.

`true` if the mutex object was locked successfully or `false` otherwise.

Description

The procedure tries to lock the mutex. `true` is returned if the mutex is locked successfully. Contrary to `vLock`, the procedure returns immediately returning `false`, if the mutex was already locked by another thread. In this case it is possible that the thread never gets the mutex if other threads are waiting for the mutex using a `vLock` call.

Remarks

Each successful call of `boTryLock` needs exactly one call of `vUnlock` within the same thread.

Example

```
{Verwendung in einem Thread}
if (oWriteMutex.boTryLock) then      {in case the mutex was locked successfully}
Writeln("Thread1");                  {prints: Thread1}
  oWriteMutex.vUnlock;               {unlock the mutex}
else                                  {unable to lock mutex}
  Writeln("No access possible");     {print a warning message}
endif;
```

toMutex**Method declaration**

```
procedure vUnlock;
```

Parameter

None.

Return value

None.

Description

A locked mutex is unlocked. Each vLock or successful call of boTryLock needs exactly one call of vUnlock.

The mutex has to be unlocked within the same thread that locked the mutex.

Remarks

None

Example

```
{using a mutex in a thread}
oWriteMutex.vLock;    {lock the mutex to access a resource}
Writeln("Thread1");  {print to standard output}
oWriteMutex.vUnlock; {unlock the mutex}
```

13 Error functions

Taken from the pool.pli library.

Introduction

Chapter 13 describes the functions to retrieve and set errors.

Section 13.2 describes the meaning of the error codes.

GetError

13.1 Functions

Declaration

```
function GetError: DWord;
```

Parameter

None.

Return value

DWord.

Error code of the last failed operation. The error code is not deleted during the function call.

Description

The function returns the error code of the last failed operation.

Remarks

A description of the error codes is provided at the end of this chapter.

A call to `GetError` after a call to a `CallFunc` function returns the error code of the library that was adjusted according to the table.

Example

```
procedure vMain;
var
  csFileNamePar: String;
  hFile:        tFile;
begin
  csFileNamePar := "";           {set the file name}
  FOpen(hFile, csFileNamePar, nenFMRead); {open file with read/write access}
  Writeln(GetError);           {prints: 2}
  Writeln(GetErrorMsg(GetError)); {prints: file not found
                                  - notice: the error code is not
                                  modified}
end;
```

SetError**Declaration**

```
procedure SetError(dwErr: DWord);
```

Parameter

DWord.

Error code to be.

Return value

None.

Description

The function sets the error code that can be retrieved using GetError.

Remarks

A description of the error codes is provided at the end of this chapter.

Example

```
procedure vMain;
begin
    SetError(10);           {set error code 8 - Out of memory }
    Writeln(GetError);     {prints: 8}
    Writeln(GetErrorMsg(GetError)); {prints: Out of memory}
end;
```

GetErrorMsg**Declaration**

```
function GetErrorMsg (dwErr: DWord): String;
```

Parameter

DWord.

Error code whose descriptive string is to be returned. To get the description of the last occurred error pass the return value of `GetError` to the `GetErrorMsg` function.

Return value

String.

Error description (if available).

Description

The function retrieves the error description of the passed error code as string. The string 'Unknown error' is returned in case no description is available

Remarks

A description of the error codes is provided at the end of this chapter.

Example

```
procedure vMain;
begin
    Writeln(GetErrorMsg(2));           {prints: No such file or directory}
    Writeln(GetErrorMsg(GetError));   {prints the description of the last
                                     Occurred error}
end;
```

Halt

Declaration

```
procedure Halt (..);
```

Parameter

```
..
```

Variable parameter list.

Return value

None.

Description

The procedure terminates POOL tasks or threads that call Halt. Optional the internal exit code is set (if passed as parameter and if the internal exit code is not bigger (see also SetExitCode). Calling Halt in the main thread branches execution to vDeinit. Calling Halt in a normal thread terminates the thread.

Remarks

Resources (especially memory used up by local strings) are not freed completely!

Mutexe und Events werden abgeräumt, aber es werden keine Konstruktoren aufgerufen.

System threads, e.g. the ones started by NoWait functions, are terminated. The operating system is not able to delete all resources of the thread. These resources are freed only when the PI terminates!

Example

```
procedure vMain;
var
  i32Num1: Int32;
  i32Num2: Int32;
begin
  repeat
    Writeln("\nEnter a number.");
    i32Num1 := I32Val(ReadStr(''));
    Writeln("\nEnter a second number.(0 terminates the program)");
    i32Num2 := I32Val(ReadStr(''));
    if (i32Num2 = 0) then
      Writeln("\nTerminate program");
      Halt;                                     {terminate the program}
    else
      Writeln("\nNumber1 / Number2 = ",i32Num1/i32Num2);
    endif;
  until false;
```

Halt

```
end;
```

SetExitCode**Declaration**

```
procedure SetExitCode(wExitCode: Word);
```

Parameter

wExitCode.

Word. Internal exit code to be set (see also GetThreadExit-Code in chapter 11 on Threads).

Possible values are:

nExitSuccess	= 0:	Thread terminated without error.
nExitWarning	= 1:	Warning.
nExitAppError1	= 2:	Application error 1.
nExitAppError2	= 3:	Application error 2 (also break).
nExitSysError	= 4:	System error (e.g. file I/O).
nExitFatalError	= 5:	Internal error.

Return value

None.

Description

The function sets the internal exit code. wExitCode overwrites the internal exit code if wExitCode = nExitSuccess or wExitCode > internal Exit-Code. Each task thread, including the main thread, has its own exit code variable.

Remarks

None.

Example

```
{within a thread}
if !FClose (fi) then           {if the file could not be closed}
  SetExit-Code(nExitWarning); {set exit code}
  return;                     {terminate thread}
endif;
```

13.2 Error codes

Introduction

The following description of the error codes is taken from the BSK-Library pool.pli and sometimes some additional explanation is provided:

Remarks

Windows reserves bit 29 to identify application error codes. GetLastError and WSAGetLastError return error codes with a cleared bit 29, that are unique too. The error codes overlap with the error codes of errno, OSAL, and AIDA. (The errno, OSAL, and AIDA errors do not overlap neither in Windows nor in Linux (at least currently)). The runtime system, resp. the entries in ErrDescr set bit 29 for Windows error codes to avoid large error code values for "normal" error codes. In case of a Linux system all errors of errno are returned which avoids to explicitly distinguish between errno errors and system errors.

Masks

Masks and constants to distinguish different error sources:

```
{ $ifdef WIN32 }      { valid until the end of standard error codes }

const
nSysErrSrcMask      = 0x20000000;
nSysErrSrcMark      = 0x20000000;
nSockErrSrcMask     = 0x20000000;
nSockErrSrcMark     = 0x20000000;
nStdErrSrcMask      = 0xFFFFFC00;
nStdErrSrcMark      = 0x00000000;
nOSALErrSrcMask     = 0xFFFFFC00;
nOSALErrSrcMark     = 0x00000400;
nAIDAErrSrcMask     = 0xFFFFFC00;
nAIDAErrSrcMark     = 0x00000800;
nADXErrSrcMask      = 0xFFFFFC00;
nADXErrSrcMark      = 0x00000C00;
```

System error codes

Attention

Currently no conversion is made with OSAL and AIDA, since the error codes are unique, they also do not collide with standard C error codes, and no Windows system error codes are returned (the conversion table in `aida.pool` is empty).

System error codes

```
const
ERROR_FILE_NOT_FOUND      =      2 | nSysErrSrcMark;
The system can not find the specified file.

ERROR_FILENAME_EXCED_RANGE= 206 | nSysErrSrcMark;
The file name or the file extension is too long.

ERROR_INVALID_HANDLE      =      6 | nSysErrSrcMark;
Invalid handle.

ERROR_INVALID_PARAMETER   =     87 | nSysErrSrcMark;
Invalid parameter.

ERROR_IO_INCOMPLETE       =   996 | nSysErrSrcMark;
Asynchronous I/O events are not signaled.

ERROR_IO_PENDING          =   997 | nSysErrSrcMark;
Asynchronous I/O events are pending.

ERROR_NOT_ENOUGH_MEMORY   =      8 | nSysErrSrcMark;
Not enough memory to perform the requested command.

ERROR_OPERATION_ABORTED   =   995 | nSysErrSrcMark;
An I/O operation was canceled because of an request of another operation or
since the thread was terminated.
```

Attention

Windows does not provide error descriptions to all possible error codes!

C error codes

Standard C error codes

(ISO-C, POSIX.1 etc., according to errno.h)

Remarks

You can get a short description of the error codes in english using the `GetErrorMsg` function. The following section gives a brief explanation of the error codes.

Detailed information can be found on the internet by searching for the function name and the string 'c', 'gnu', and 'library' (see example).

Example



Bild 2: Searching for an description of the error ENOLCK.

```
const
E2BIG          = 7 | nStdErrSrcMark;
Argument list is too long.

EACCES        = 13 | nStdErrSrcMark;
Access denied.

EAGAIN        = 11 | nStdErrSrcMark;
Returned if a resource (memory, process or nested level) is currently not
available. The same function call can succeed if issued a second time after a
while.

EBADF         = 9 | nStdErrSrcMark;
Invalid File handle.

EBUSY        = 16 | nStdErrSrcMark;
Device or resource is busy.

ECHILD       = 10 | nStdErrSrcMark;
No child process available.

ECHRNG       = 44 | nStdErrSrcMark;
Die Nummer des Kanals befindet sich außerhalb des zulässigen Bereichs.

EDEADLK      = 36 | nStdErrSrcMark;
Execution of the command would lead to a resource deadlock.
```

C error codes

EDOM	= 33	nStdErrSrcMark;	The argument of a mathematical function is out of the valid range.
EEXIST	= 17	nStdErrSrcMark;	File exists.
EFAULT	= 14	nStdErrSrcMark;	Invalid address.
EFBIG	= 27	nStdErrSrcMark;	File is to big.
EILSEQ	= 42	nStdErrSrcMark;	Illegale byte order within a "multibyte" or "wide character".
EINTR	= 4	nStdErrSrcMark;	Interrupted function or system call.
EINVAL	= 22	nStdErrSrcMark;	Invalid parameter. In POOL the error occurs among other things when a calling the constructor on a previously initialized object.
EIO	= 5	nStdErrSrcMark;	I/O error.
EISDIR	= 21	nStdErrSrcMark;	Error while trying to write to a directory.
EMFILE	= 24	nStdErrSrcMark;	To many files opened.
EMLINK	= 31	nStdErrSrcMark;	To many connections.
ENAMETOOLONG	= 38	nStdErrSrcMark;	File name too long.
ENFILE	= 23	nStdErrSrcMark;	The system has to many open files. File table overrun.
ENODATA	= 61	nStdErrSrcMark;	No data available.
ENODEV	= 19	nStdErrSrcMark;	No device.
ENOENT	= 2	nStdErrSrcMark;	File or directory does not exist.
ENOEXEC	= 8	nStdErrSrcMark;	File is not executabe (wrong format).
ENOLCK	= 39	nStdErrSrcMark;	The system has run out of file pointers.
ENOMEM	= 12	nStdErrSrcMark;	No enough memory.

OSAL-Error codes

```

ENOSPC          = 28 | nStdErrSrcMark;
Out of memory.

ENOSYS          = 40 | nStdErrSrcMark;
Function is not implemented.

ENOTDIR         = 20 | nStdErrSrcMark;
Passed parameter is not a directory. A directory is required.

ENOTEMPTY       = 41 | nStdErrSrcMark;
Directory is not empty.

ENOTTY         = 25 | nStdErrSrcMark;
Not a output unit.

ENXIO          = 6 | nStdErrSrcMark;
Device or address not available.

EPERM          = 1 | nStdErrSrcMark;
Operation not allowed.

EPIPE         = 32 | nStdErrSrcMark;
Connection interrupted.

ERANGE        = 34 | nStdErrSrcMark;
Unable to display the result of a mathematical operation.

EROFS         = 30 | nStdErrSrcMark;
Write access not possible. Read only file system.

#ESPIPE       = 29 | nStdErrSrcMark;
The file pointer is not allowed to point to the specified connction or FIFO.

ESRCH        = 3 | nStdErrSrcMark;
No process exists that has the passed ID.

EXDEV        = 18 | nStdErrSrcMark;
Attempt to create an invalid connection between file systems.
(1) Supported by PI and Linux. Not defined in Windows.

{$else}      // WIN32 not defined
  error 'Error constants not defined yet!'
{$endif}    // Ende of $ifdef WIN32

```

Standard-BSK-OSAL-Error codes

```

const
EOSAL_CANT_ACCESS_FILE = 8 | nOSALErrSrcMark;
File not found, not accessible, etc.

EOSAL_FATAL            = 0 | nOSALErrSrcMark;
Fatal error (memory structures destroyed).

```

OSAL-Error codes

```
EOSAL_INIT_FAILED          = 12 | nOSALErrSrcMark;  
Initialisation error.  
  
EOSAL_INVALID_DATA        = 10 | nOSALErrSrcMark;  
Invalid value.  
  
EOSAL_INVALID_HANDLE      =  3 | nOSALErrSrcMark;  
Invalid OSAL handle.  
  
EOSAL_INVALID_MODULE      = 11 | nOSALErrSrcMark;  
Not a valid module.  
  
EOSAL_INVALID_PARAMETER   =  4 | nOSALErrSrcMark;  
Invalid parameter.  
  
EOSAL_LOCK                 =  6 | nOSALErrSrcMark;  
Lock failed.  
  
EOSAL_NOT_ENOUGH_MEMORY   =  2 | nOSALErrSrcMark;  
Not enough memory.  
  
EOSAL_NOT_FOUND           =  5 | nOSALErrSrcMark;  
File, module, or object not found.  
  
EOSAL_OBJECT_ALREADY_EXISTS =  7 | nOSALErrSrcMark;  
Object exists.  
  
EOSAL_QUEUE_FULL          = 13 | nOSALErrSrcMark;  
Queue congestion. Can not queue additional elements.  
  
EOSAL_UNSPECIFIED         =  9 | nOSALErrSrcMark;  
Unspecified error.  
  
EOSAL_CALL_NOT_IMPLEMENTED =  1 | nOSALErrSrcMark;  
Function is not supported.
```

14 System functions

14.1 General system functions

Introduction

You can use the system functions to set and get system environment variables, normalize a path, and to start external programs.

ExpEnvVars

Declaration

```
function ExpEnvVars(cs: String): String;
```

Parameter

cs.

String. String containing the name of the environment variable. Syntax: \$(name)

Return value

String.

String containing the path of the environment variable or an empty string if the environment variable was not found.

Description

The name of an environment variable is passed to the function. The function extracts the path and return it as string.

Expandiert wird dabei:

\$(NAME) -> NAME extracted from the environment variable resp. "" (empty string) if the environment variable was not found.

\$(NAME/) -> same as above plus additional „/" if NAME is not empty and the content does not end with : resp. / (whereby not the expanded content is evaluated, with NAME=\$(NAME2) and NAME2=/test/ is \$(NAME/) expanded to /test/).

Remarks

The names are possibly case sensitive.

"\$\$" is transformed to "\$" and not further processed, e.g. "\$\$(NAME)" is converted to "\$(NAME)", without substituting NAME with its environment variable value.

The environment variable can contain other environment variables that get expanded too.

The environment variable names itself can not contain environment variables, i.e. \$(NAME\$(SUFFIX)) is not allowed.

cs can contain multiple environment variables.

Processing stops with the occurrence of a '\0' regardless of the real length of cs.

The name of the environment variable is not allowed to end with '\' (whereas the content is allowed to).

AIDABIN is set to the PI.EXE path if not set in the environment to ensure that AIDABIN exists always. AIDABIN is normalized in any case and a '/' is attached if necessary.

ExpEnvVars

The environment variable CWD (current working directory) always contains the current working directory, even if CWD is set to another value in the environment settings.

\$(CWD) and \$(AIDABIN) are useful to create an absolute path name e.g. for creating parameters for VObj.

Example

(see also example of GetEnv)

```
procedure vMain;
begin
  Writeln(ExpEnvVars("$(CWD)")); // print current working directory
  Writeln(ExpEnvVars("$(AIDABIN)")); // print AIDABIN path
end;
```

GetEnv**Declaration**

```
function GetEnv(csName: String): String;
```

Parameter

cs.

String. A string containing the name of the environment variable.

Return value

String.

A string containing the value of the environment variable or an empty string if the environment variable does not exist.

Description

The function returns the value of the environment variable.

Remarks

The function does not expand environment variables that are contained within the requested environment variable, contrary to ExpEnvVars.

Example

```
procedure vMain;
begin
  Writeln(GetEnv("TESTVAR1"));           {prints: C:\programs\bsk}
  Writeln(ExpEnvVars("${TESTVAR2}"));   {prints: C:\Programs\bsk\test}
  Writeln(GetEnv("AIDABIN5"));         {prints: ${TESTVAR1}\test}
end;
```

PutEnv**Declaration**

```
function PutEnv(csName, csVal: String): Boolean; ifr;
```

Parameter

csName.

String. Name of the environment variable.

csVal.

String. Value of the environment variable or an empty string to "delete" the environment variable.

Return value

Boolean.

true if successful. false in case of an error (name not found).

Description

The function set the environment variable that is specified in the csName parameter to the value provided with the csVal parameter. The environment variable is created if it does not exist. The modification (includes deletion) is not permanent, but limited to the runtime of the program that modified the environment variables.

Remarks

The environment variables are global to all POOL-Tasks and libraries that are loaded by the PI. The modified environment variables are not valid within the Commander and VObj, since they are started before the PI.

The handling of names depends on the system (allowed characters, case sensitivity etc.). You should only use upper case letters, numbers and the underscore for system compatibility.

The special charactes (\$, (,), /, \, : and ;) are prohibited because of internal special treatment.

The modifier `ifr` indicates that you can ignore the return value.

Example

```
procedure vMain;
begin
  Writeln(GetEnv("AIDABIN5")); {prints: path of the enviroment variable}
  Writeln(PutEnv("AIDABIN5", "C:\\programs"));
```

PutEnv

```
Writeln(GetEnv("AIDABIN5")); {prints: C:\programs}
end;
```

NormPath

Declaration

```
function NormPath(var csPath: String; boName: Boolean): tenNPREs; ifr;
```

Parameter

csPath.

String as reference. Path to normalize (including file name if necessary).

boName.

Boolean.

boName indicates, whether csPath is allowed to end with a name (a path separator character is attached (if necessary) if boName is `false`).

Return value

tenNPREs.

Error respective status code ordered by the severity of the error.

nenNPOk:	Result is okay, without "../".
nenNPUp:	Result starts with "../" (dependent of CWD).
nenNPDriveUp:	Result starts with "D:../" (dependent CWD(D)).
nenNPHostUp:	Result starts with "//Host/Vol/.." (normalisation impossible).
nenNPRootUp:	Result starts with "/../" or "D:../" (normalisation impossible, if path is to be absolute).
nenNPOverrun:	No space to attach '/' or expand e.g. "/../" within FILENAME_MAX (i.e. no complete result).

Description

The function normalizes the passed path. boName indicates, whether csPath is allowed to end with a name (a path separator character is attached (if necessary) if boName is `false`).

Remarks

Leading and trailing whitespaces are removed and all other whitespaces are replaced by a blank.

NormPath

'\' is replaced by '/' on DOS/Windows machines. '/' is appended if `boName` is `false`, and `csPath` is not empty, and the last character is not '/' (or ':' on DOS/Windows machines).

"/" (except at the beginning) and "/." are converted to "/" (Additionally ":\." is converted to ":" on DOS/Windows machines)

"/" at the beginning sustains the first two names (host and e.g. volume name) under any circumstances.

"/.." etc. is replaced by removing directories if possible (e.g. "a/b/..c" is replaced by "a/c").

In case that "/.." etc. are left over, then these are replaced by the appropriate number of "/..".

Neither case conversion takes place (to make names unique use `LoStr` on DOS/Windows machines) nor a check for illegale characters.

The modifier `ifr` indicates that you can ignore the return value.

Example

```
procedure vMain;
var
  csPath: String;
  enNPres: tenNPres;
begin
  csPath := "C:\\programs\\bsk";
  NormPath(csPath, false);
  Writeln(csPath);    {prints: C:/programs/bsk/}
  Writeln(enNPres);
end;
```

Path2Host**Declaration**

```
function Path2Host(csPath: String): String;
```

Parameter

csPath.

String. A path that is to be converted into host format.

Return value

String.

A string containing csPath converted into host format.

Description

The function converts the passed path into host format.

Remarks

Currently only used by the system to convert DOS/Windows path separators. A path containing blanks has to be enclosed by double quotes. A single DOS/Windows path separator is doubled at the end to allow this without running into problems.

Example

```
procedure vMain;
var
  csPath:      String;
  csHostPath: String;

begin
  csPath := "C:/programs/bsk";
  csHostPath := Path2Host(csPath);
  Writeln(csHostPath);    {prints (WinNT): C:\programs\bsk}
end;
```

System

Declaration

```
function System (cs: String, dw: DWord): Int32;
```

Parameter

csString.

String. Path and program name (normalized using Path2Host if necessary).

dw.

DWord. Options for opening the program.

Possible values:

nSFNoConMask = 0x00000001

Do not open a console for a console program (ignored in case of an Windows application, or an NTVDM error message in case of an 16-Bit application). Pass 0 to simply start the program.

Return value

Int32.

Result: -1: Error when calling program. External program was not started or empty command was specified.

Result: >=0: Exit code of the called program.

Description

The function starts an external program. Program path and name has to be specified. You can specify additional start parameters using dw

Remarks

The function is able to start links.

The complete command line is specific to the underlying system. Therefore it can be necessary to use the function Path2Host at least for the program name.

DOS/Windows command.com does not return the exit code of the called program. Therefore the command line is interpreted within the runtime system and the program is launched directly in an DOS/Windows environment. In case that this differences disturb or if internal functions of the command processor or Input/Output redirection is needed you have to call Command or Cmd using ExpEnvVars ("\$(COMSPEC) /C ..").

cmd.exe is used on Win/NT and 2000 systems instead of command.com for starting 32-Bit programs, since the exit code is returned correctly even from a batch file.

System**Example**

```
procedure vMain;
var
  dwTest: DWord;

begin
  dwTest := 0;
  Writeln(System("c:\\programs\\ActiveX_Test.exe", dwTest));
end;
```

System_NW

Declaration

```
procedure System_NW (var oNWEvent: toNWEvent; cs: String;dw:DWord);
```

Parameter

oNWEvent.

toNWEvent as referece (see also toNWEvent and toEvent). The object contains the error code (oNWEvent.dwErrNo) after the event is signaled and possibly a result of the program call (oNWEvent.unVal.dw, oNWEvent.unVal.i32, or oNWEvent.unVal.pv). For details see the description of the toEvent object.

csString.

String. Path and program name (normalized using Path2Host if necessary).

dw.

DWord. Options for opening the program.

Possible values:

nSFNoConMask = 0x00000001

Do not open a console for a console program (ignored in case of an Windows applicationen, or an NTVDM error message in case of an 16-Bit application). Pass 0 to simply start the program.

Return value

None.

Description

The function starts an external program. Program path and name has to be specified. Contrary to the function System the function System_NW does not wait until the program has started. Of course you can wait for an event if necessary using WaitEvent(s).

Possible return values of the program can the retrieved using oNWEvent.unVal.dw, oNWEvent.unVal.i32 or oNWEvent.unVal.pv and oNWEvent.dwErrNo contains the error code if any is provided.

Remarks

The function is able to start links.

The complete command line is specific to the underlying system. Therefore it can be necessary to use the function Path2Host at least for the program name.

System_NW

DOS/Windows's Command does not return the exit code of the called program. Therefore the command line is interpreted within the runtime system and the program is launched directly in an DOS/Windows environment. In case that this differences disturb or if internal functions of the command processor or Input/Output redirection is needed you have to call Command or Cmd using ExpEnvVars ("\$(COMSPEC) /C ..").

cmd.exe is used on Win/NT and 2000 systems instead of command.com for starting 32-Bit programs, since the exit code is returned correctly even from a batch file

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (ReadKey, WaitKey, GetLine, MenuExecute, IDECommand, or the appropriate NoWait versions).

Example

(Calling a program with a timeout of 5s)

```
module Test;
private
var
  oNWEvent: toNWEvent;
  oTEvent:  toTEvent;

procedure vMain;
var
  dwTest:  DWord;
  oNWEvent: toNWEvent;

begin
  {start the program}
  System_NW(oNWEvent, "c:\\programs\\ActiveX_Test.exe", 0);
  {wait for an event}
  if WaitEvents(oNWEvent, oTEvent) = tpoEvent(@oNWEvent) then
    Writeln("Program was started successfully");
    oTEvent.vSignal;
  else
    Writeln("Could not start program");
  endif;
end;

procedure vDeinit;
begin
  oNWEvent.vDone;
  oTEvent.vDone;
end;

begin
  oTEvent.poInit(5000);
  oNWEvent.poInit;
end.
```

14.2 Date and time functions

Declaration

```
function GetTime_ms: DWord;
```

Parameter

None.

Return value

DWord.

Time in milli seconds since the system or task was started.

Description

The function returns the time in milli seconds since the system or task was started.

Remarks

You should always use differences when calculating since the time starts at zero again after approx. 49,7 days (counter overrun).

Example

```
procedure vMain;
var
  dwHelp: DWord;

begin
  dwHelp := GetTime_ms;           // get current time
  Wait(5000);
  Writeln(GetTime_ms - dwHelp);  // print the elapsed time since first call
end;
```

GetTime_s**Declaration**

```
function GetTime_s: DWord;
```

Parameter

None.

Return value

DWord.

Elapsed time in seconds since 01.01.1970 at 00:00:00Z.

Description

The function returns the elapsed time in seconds since 01.01.1970 at 00:00:00Z.

Remarks

DWORD_MAX is returned in case of an error.

The C runtime library currently returns INT32_MAX=19.01.2038 03:14:07Z.

Example

```
procedure vMain;
var
  dwHelp: DWord;

begin
  dwHelp := GetTime_s;           // get current time
  Wait(5000);
  Writeln(GetTime_s - dwHelp); // print the elapsed time since first call
end;
```

MakeDateTime

Declaration

```
procedure MakeDateTime(var : tstDateTime; dwTime_s: DWord; enTZ: tenTZ);
```

Parameter

stDateTime.

tstDateTime as reference. Structure, that receives the result (see also the description of the structure tstDateTime).

dwTime_s.

DWord. A time that is to be written into the DateTime structure. Pass the GetTime_s result to the function in order to get the current time.

enTZ.

tenTZ. Desired time zone.

Possible values are:

nenTZUTC: Coordinated Universal Time. (The international time standard).

nenTZLocal: Local time.

Return value

None.

Description

The procedure converts the passed time (in seconds) into a DateTime structure (see also the description of the structure tstDateTime). The structure is passed to the procedure as reference. The desired time zone is selected via enTZ. You can choose between UTC and local time.

Remarks

A reseted record is returned in case of an error.

Notice: The C runtime library currently returns INT32_MAX=2038-01-19 03:14:07Z.

Example

```
procedure vMain;
var
  stDateTime: tstDateTime;
```

MakeDateTime

```
begin
  MakeDateTime (stDateTime, GetTime_s, nenTZLocal);
  Writeln (stDateTime);
end;
```

MakeTime_s**Declaration**

```
function MakeTime_s(var stDateTime: tstDateTime): DWord;
```

Parameter

stDateTime.

tstDateTime as reference. A structure that contains a time and date.

Return value

DWord.

The number of elapsed seconds since 01.01.1970 at 00:00:00Z.

Description

The function returns the number of elapsed seconds since 01.01.1970 at 00:00:00Z.

Remarks

The fields wYear, bSec, i8IsDst, and enTZ have to be set, for UTC i8IsDst=0, for local time -1 (thus the function determines whether to use Daylight Savings Time if possible) aktiv ist.

The rest of the fields are set if the calculation is successful. The fields wYear and bSec are normalized (e.g. 32.Jan → 1.Feb).

DWORD_MAX is returned in case of an error

Notice: The C runtime library currently returns INT32_MAX=2038-01-19 03:14:07Z.

Example

```
procedure vMain;
var
  stDateTime: tstDateTime;
  dwTime_s:   DWord;

begin

  dwTime_s := GetTime_s;   {get current time}
  Writeln(dwTime_s);

  {convert time in DateTime structure}
  MakeDateTime(stDateTime, dwTime_s, nenTZLocal);

  {convert and print time from DateTime structure}
  Writeln(MakeTime_s(stDateTime));

end;
```

15 General library functions

Introduction

You can use the POOL library functions to call external library functions (e.h. DLL functions) from a POOL application. You have to ensure that you do not introduce system dependencies. This is the reason why you should not use system dlls in your POOL applications.

LoadLib

Declaration

```
function LoadLib (csLibName: String): tpstLibDescr;
```

Parameter

csLibName.

String. Name (and if necessary the path) of the library (DLL).

Return value

tpstLibDescr.

If successful a structure is returned that contains the library name, a handle to the library, and a pointer to the error structure (see also the description of the structure tpstLibDescr). A nil pointer is returned in case of an error.

Description

LoadLib loads the specified library (DLL) using its name. A structure containing information on the library is returned in case of success, or nil otherwise.

Remarks

None.

Example

```
procedure vMain;
var
  stLFDescr: tstLFDescr;
  dwTest:    DWord;
begin
  stLFDescr.pstLibDescr := LoadLib('User32'); {load the library}

  {in case the library could not be loaded}
  if stLFDescr.pstLibDescr = nil then
    Writeln('Error: ',GetErrorMsg(GetError),' (User32)');
    return;
  endif;

  {retrieve the address of the function MessageBoxA}
  if not GetLibFunction(stLFDescr, stLFDescr.pstLibDescr, 'MessageBoxA') then
    Writeln('MessageBoxA not found in User32 -',GetErrorMsg(GetError));
    return;
  endif;
```

LoadLib

```
repeat
  {call the function and pass parameters}
  dwTest := LibCall (stLFDescr,nil,"Do you want to exit the program?",
                    "Warning",1);
  if dwTest = 1 then
    WritelnStr('Exit program');
  elseif dwTest = 2 then
    WritelnStr('Resume program');
  else
    WritelnStr('Error - resuming program');
  endif;
  Wait(5000);
until dwTest = 1
{unload the library}
UnloadLib(stLFDescr.pstLibDescr);
end;
```

Attention

System specific library functions are only used to demonstrate the use of LibCall. You should not use system specific library functions in your AIDA applications since you would give up portability!

Output to a remote monitor (in case PI and Commander are running on different machines) are useless in most cases.

The message box is not displayed using MessageBoxA if the PI was started without an own window! For this example, core.run has to be started with parameter 2 instead of the usual parameter 0.

The remarks were taken from the example program win_msg.pool by BSK that also uses the MessageBoxA function.

UnloadLib

Declaration

```
function UnloadLib (var pstLibDescr: tpstLibDescr): Boolean; ifr;
```

Parameter

pstLibDescr.

tpstLibDescr as reference. A structure, that contains a handle to a library that was loaded using LoadLib. This handle is used to identify the library.

Return value

Boolean.

true, if the library was unloaded successfully, false otherwise.

Description

The function unloads the library that is referenced by the handle and deletes the pointer to the structure. true is returned if the library was unloaded successfully, false otherwise.

Remarks

The pointer to the structure is deleted in any case when calling UnloadLib.

The modifier ifr indicates that you can ignore the return value.

Example

(see also the example to LoadLib);

```
{unload the library}
if !(UnloadLib(stLFDescr.pstLibDescr)) then
  Writeln(GetErrorMsg(GetError));
  return;
endif;
```

GetLibFunction

Declaration

```
function GetLibFunction (var stDescr: tstLFDescr; pstLibDescr: tpstLibDescr;  
csFuncName: String): Boolean;
```

Parameter

stDescr.

tstLFDescr as reference. A structure, that receives the name and a pointer to the function whose name was passed via csFuncName and a pointer to a structure (pstLibDescr) containing the handle to the library, if successful.

pstLibDescr.

tpstLibDescr. A structure, that contains a handle to library that was loaded using LoadLib. This handle is used to identify the library.

csFuncName.

String. The name of the function, that is to be referenced via stDescr. You can pass an empty string, if the name is set in stDescr.

Return value

Boolean.

true, if the function could be found, otherwise false.

Description

The function retrieves the address of the function (csFuncName) contained in the library identified by the passed library handle. You can pass an empty string, if the name is set in stDescr.

The address of the function, the function name and a pointer to the structure that contains the library handle is returned via the stDescr parameter.

true is returned if successful, otherwise false.

Remarks

None.

Example

(see also the example to the LoadLib function);

GetLibFunction

```
{retrieve the address to the MessageBoxA function}
if not GetLibFunction(stLFDescr, stLFDescr.pstLibDescr, 'MessageBoxA') then
    Writeln('MessageBoxA not found in User32 -',GetErrorMsg(GetError));
    return;
endif;
```

LibCall**Declaration**

```
function LibCall (var stDescr: tstLFDescr; ..): DWord;
```

Parameter

stDescr.

tstLFDescr as reference. A structure that contains a pointer to the function to call, the function name and a pointer to a structure (pstLibDescr) containing the library handle.

..

Variable parameterlist. Parameters for the library function.

Variants of the LibCall function with different parameters**Declaration**

```
function LibCall (pstLibDescr: tpstLibDescr; csFuncName: String;..): DWord;
```

(see remarks)

Parameter

pstLibDescr.

tpstLibDescr. A structure that contains the handle to identify a library, loaded using LoadLib.

csFuncName.

String. Name of the function that is to be called.

Declaration

```
function LibCall (csLibName: String; csFuncName: String; ..): DWord;
```

(see remarks)

Parameter

csLibName.

String. Name (and path if necessary) of the library (DLL).

csFuncName.

String. Name of the function that is to be called.

LibCall**Declaration**

```
function LibCall (..): DWord;
```

(see remarks)

Parameter

..

Variable parameter list.

Return value

DWord.

Function result. Depends on the library function.

Description

The function LibCall calls an external function using its address that was retrieved during a previous call to the GetLib function. The library was loaded using LoadLib and is identified by a handle (stored in the structure stLFDscr). The LibCall result is the result of the external function (if the function was called and executed) and a possibly internal error code (see SetLibErrDscr). In case of invalid parameters or if the function was not found, 0 is returned with an appropriate set internal error code. The interpretation of the function result is done within the application (see also the remarks).

Remarks

Additional variants of the function LibCall including short descriptions are provided following to the example.

Use only functions that need only some milli seconds to execute since all POOL tasks and threads are stopped during the function call! Use the function LicCall_NW for functions that need longer to execute!

The application has to "know" how to interpret the function result (e.g. as Int32 or as pointer to the actual return value).

You have to provide the parameter exactly the way the function expects them. For the actual pass action the parameters are widened to multiple of four bytes and (Byte)Strings are passed as pointers. Empty char strings are passed as pointers to acEmptyHStr, empty ByteStrings are passed als nil pointer. You have to explicitly specify the length of ByteStrings in bytes, in case the length is needed. Char is always passed a string (since POOL does not distinguish syntactically between char constants and string constants). You can use Ord(c) to pass characters if necessary.

The support of calls using the name is intended for interactive usage. Use tstFLDscr in you applicationszu, also for error handling!

LibCall**Example**

(see also the example the the LoadLib function);

```
dwTest := LibCall (stLFDdescr,nil," Do you want to exit the program?",
                  "Warning",1);
if dwTest = 1 then
  WritelnStr('Exit program');
elseif dwTest = 2 then
  WritelnStr('Resume program');
else
  WritelnStr('Error - resuming program');
endif;
```

LibCall_NW**Declaration**

```
procedure LibCall_NW (var oNWEvent: toNWEvent; var stDescr: tstLFDescr; ..);
```

Parameter

oNWEvent.

`toNWEvent` as reference (see also `toNWEvent` and `toEvent`). The object contains the error code (`oNWEvent.dwErrNo`) after the event is signaled and possibly a result of the function call (`oNWEvent.unVal.dw`, `oNWEvent.unVal.i32`, or `oNWEvent.unVal.pv`). For details see the description of the `toEvent` object.

stDescr.

`tstLFDescr` as reference. A structure, that contains a pointer to the function to call, the function name and a pointer to the structure containing the library handle.

..

Variable parameter list. Parameters for the library function.

Return value

None.

Description

The procedure `LibCall` calls an external function using its address that was retrieved during a previous call to the `GetLib` function. The library was loaded using `LoadLib` and is identified by a handle (stored in the structure `stLFDescr`). The result of the call to the external function can be retrieved using `oNWEvent.unVal.dw`, `oNWEvent.unVal.i32` or `oNWEvent.unVal.pv` (see the example of `toEvent`).

The functions `WaitEvent(s)` can be used to wait for the event. You can also signal the event manually using `vSignal`.

You can retrieve a possibly set internal error code using a pointer to the structure `stLibDescr` (see `SetLibErrDescr`). The internal error code were set if invalid parameters were passed or the function was not found. The interpretation of the result is done in the application (see also remarks).

Remarks

Since it is currently only possible to work with one Commander window and the Commander does not process input and output functions in parallel, the entire communication for the input and output window is blocked by any function that waits for Commander data (`ReadKey`, `WaitKey`, `GetLine`, `MenuExecute`, `IDECommand`, or the appropriate `NoWait` versions).

LibCall_NW

You are not allowed to modify the content of pointers or strings that are passed as parameter, until the external function has completed i.e. the event is signaled!

The procedure LibCall_NW proceeds immediately without waiting for the result of the external function like the LibCall function. You should use this function when the called function possibly needs to much time to execute to wait for (see also LibCall).

Example

```

module Test;

private
var
  stLFDescr: tstLFDescr;

  oNWEvent:  toNWEvent;
  oTEvent:   toTEvent;
  poEvent:   tpoEvent;

procedure vLibFunc;
var
  dwVal: DWord;

begin
  {load the library}
  stLFDescr.pstLibDescr := LoadLib('User32');
  if stLFDescr.pstLibDescr = nil then
    Writeln('Error: ',GetErrorMsg(GetError), ' (User32)');
    return;
  endif;
  {retrieve the address of the function }
  if not GetLibFunction(stLFDescr, stLFDescr.pstLibDescr, 'MessageBoxA') then
    Writeln('Function or library not found - ', GetErrorMsg(GetError));
    return;
  endif;

  repeat
    {call the function without waiting for the result }
    LibCall_NW(oNWEvent,stLFDescr, nil,"Terminate program?","Warning", 1);
    repeat
      poEvent := WaitEvents(oNWEvent,oTEvent);
      Write('.');
      if poEvent = nil then
        oNWEvent.dwErrNo := GetError;
        break;
      endif;
    until poEvent=Addr(oNWEvent);
    if oNWEvent.dwErrNo<>0 then
      Writeln('Error: ',GetErrorMsg(oNWEvent.dwErrNo));
    else
      dwVal := oNWEvent.unVal.dw;      {determine function result}
      if dwVal=1 then
        WritelnStr('Program terminates.');
```

LibCall_NW

```
        WritelnStr('Program continues');
    else
        WritelnStr('Error');
    endif;
endif;
Wait(2000);
until dwVal = 1;           {until OK button is pressed}
end;

procedure vMain;
begin
    vLibFunc;             {call the function}
end;

procedure vDeninit;
begin
    UnloadLib(stLFDdescr.pstLibDescr);
    oNWEvent.vDone;
    oTEvent.vDone;
end;

begin
    oNWEvent.poInit;
    oTEvent.poInit(1000);
end.
```

CallFunc

Declaration

```
function CallFunc (pFuncAdr: Pointer; pstErrDescr: tpstErrDescr; pParameter: Pointer; xoSize: tSize): DWord;
```

Parameter

pFuncAdr.

Pointer. Address of the library function. The address is retrieved using the GetLibfunction and returned via the stLFDescr structure (stLFDescr.pFuncAddr).

pstErrDescr.

tpstErrDescr. Pointer to a structure that is used to administer the errors of Lib and NoWait functions. You have to initialize the structure with the error handling functions of the dll and a table to translate the error codes (see the example and the description of the SetLibErrDescr function).

pParameter.

Pointer. A pointer to the parameters of the library function. Create a structure and pass a pointer to the structure to pass multiple parameters. The order of the parameters has to conform to the order as the function expects them.

xoSize.

tSize equals Int32. Size of the passed parameters. You can retrieve the size using SizeOf(Parameter), whereas „Parameter“ is the structure to pass.

Return value

DWord.

Return value of the library function. You can get a possible error code using GetError.

Description

CallFunc calls a function using its address and passes the parameters. Contrary to LibCall, CallFunc provides addresses to the error handling functions (located in the dll) to set and get the error. Use the function GetLibFunction to get the function's address. The result of the CallFunc function is the result of the external function. The result is to be interpreted within the application (e.g. as Int32 or as pointer to the actual return value).

You have to pass the parameters via pParameter in the order and as type as they are expected. You can do this by creating a structure containing the parameters and passing a pointer to this structure. You also have to specify the size of the parameters.

CallFunc

You can determine the size using the SizeOf function. The functions to do the error handling and the table to translate the error codes are passed using a pointer to a tstErrDescr structure which you have to initialize. Afterwards you can get the error code using the GetError function.

Remarks

Use the CallFunc_NW function instead of the CallFunc function to call functions that need more than some milli seconds to execute, since all POOL tasks and thread are blocked until the CallFunc function executes!

When you call the CallFunc, the function to reset the error code is called automatically if available using pSetErrorFunc. After this the function call to the desired function of the dll is executed followed by a call to retrieve the error code using pGetErrorFunc. The error code is 0 if no error occurred.

Example

```

module Test;

{ structure containing the parameters (just one in this example) }
type
  tstPar = record
    i32Value: Int32;
  end;

private

procedure vMain;
var
  stLFDescr:  tstLFDescr;      {structure containing the function description}
  stLFSetErr: tstLFDescr;     {structure for the error function SetError}
  stLFGetErr: tstLFDescr;     {structure for the error function GetError}
  stErrDescr: tstErrDescr;    {error structure}
  dwRes:      DWord;
  stPar:      tstPar;         {structure containing the parameters}

const
  {table to translate error values}
  adwSockMMO: array[0..1*3] of DWord = (1, DWORD_MAX,100,1);

begin

  stLFDescr.pstLibDescr := LoadLib('TestDLL');

  {in case the library could not be loaded}
  if stLFDescr.pstLibDescr = nil then
    Writeln('Error: Could not load library - ',GetErrorMsg(GetError));
    return;
  endif;

  {assign the library to the error function structures}
  stLFSetErr.pstLibDescr := stLFDescr.pstLibDescr;
  stLFGetErr.pstLibDescr := stLFDescr.pstLibDescr;

```

CallFunc

```

{retrieve the address of the TESTDLL_i32GetValue function}
if not GetLibFunction(stLFDescr, stLFDescr.pstLibDescr,
                    'TESTDLL_i32GetValue') then
    Writeln("Function i32GetValue not found - ", GetErrorMsg(GetError));
    return;
endif;

{retrieve the address of the TESTDLL_dwGetError function}
if not GetLibFunction(stLFGetErr, stLFGetErr.pstLibDescr,
                    'TESTDLL_dwGetError') then
    Writeln("Function dwGetError not found ", GetErrorMsg(GetError));
    return;
endif;

{retrieve the address of the TESTDLL_vSetError function}
if not GetLibFunction(stLFSetErr, stLFSetErr.pstLibDescr,
                    'TESTDLL_vSetError') then
    Writeln("Function vSetError not found ", GetErrorMsg(GetError));
    return;
endif;

{assign the error functions to the error structures}
stErrDescr.pSetErrorFunc := stLFSetErr.pFuncAddr;
stErrDescr.pGetErrorFunc := stLFGetErr.pFuncAddr;

{assign the translation table}
stErrDescr.padwMMO := tpaDWord(@adwSockMMO);

{assign the error structure to the function structure}
SetLibErrDescr(stLFDescr.pstLibDescr, @stErrDescr);

{call the external function (this calls the error functions automatically)}
stPar.i32Value := 150;
dwRes := CallFunc(stLFDescr.pFuncAddr, @stErrDescr, @stPar, SizeOf(stPar));
Writeln(dwRes);
                    {prints: 300 (DLL function doubles the
                    passed value}

Writeln(GetErrorMsg(GetError)); {print the error message}

UnloadLib(stLFDescr.pstLibDescr);
end;

procedure vDeinit;
begin
end;

begin
end.

```

CallFunc_NW

Declaration

```
procedure CallFunc_NW (var oNWEvent: tONWEvent;  
    pFuncAdr: Pointer; pstErrDescr: tpstErrDescr;  
    pParameter: Pointer; xoSize: tSize);
```

Parameter

oNWEvent.

`tONWEvent` as reference. Event class of the NoWait objects. The object has to be initialized using the constructor `polnit` before it is used and it has to be deleted using `vDone` before the program terminates.

The object contains the error code (`oNWEvent.dwErrNo`) after the event is signaled and possibly the result of the function call (`oNWEvent.unVal.dw`, `oNWEvent.unVal.i32`, or `oNWEvent.unVal.pv`). For details see the description of the `tOEvent` object.

pFuncAdr.

`Pointer`. Address of the library function. The address is retrieved using the `GetLibfunction` and returned via the `stLFDescr` structure (`stLFDescr.pFuncAddr`).

pstErrDescr.

`tpstErrDescr`. Pointer to a structure that is used to administer the errors of Lib and NoWait functions. You have to initialize the structure with the error handling functions of the dll and a table to translate the error codes (see the example and the description of the `SetLibErrDescr` function).

pParameter.

`Pointer`. A pointer to the parameters of the library function. Create a structure and pass a pointer to the structure to pass multiple parameters. The order of the parameters has to conform to the order as the function expects them.

xoSize.

`tSize` equals `Int32`. Size of the passed parameters. You can retrieve the size using `SizeOf(Parameter)`, whereas „Parameter“ is the structure to pass.

Return value

None.

CallFunc_NW

Description

CallFunc_NW calls a function using its address and passes the parameters. Use the function GetLibFunction to get the function's address. Use a NoWait event and the WaitEvent(s) functions to get the function's result (oNWEvent.unVal.i32, oNWEvent.unVal.dw or oNWEvent.unVal.pv). The result is to be interpreted within the application (e.g. as Int32 or as pointer to the actual return value).

Contrary to LibCall, CallFunc provides addresses to the functions (located in the dll) to set and get the error.

You have to pass the parameters via pParameter in the order and as type as they are expected. You can do this by creating a structure containing the parameters and passing a pointer to this structure. You also have to specify the size of the parameters. You can determine the size using the SizeOf function. The functions to do the error handling and the table to translate the error codes are passed using a pointer to a tstErrDescr structure which you have to initialize. Afterwards you can get the error code using the GetError function.

Remarks

Use the CallFunc_NW function instead of the CallFunc function to call functions that need more than some milli seconds to execute, since all POOL tasks and thread are blocked until the CallFunc function executes.

The structure pstErrDescr contains a pointer to the library's error handling functions and an array to translate error codes. When you call the CallFunc_NW, the function to reset the error code is called automatically if available using pSetErrorFunc. After this the function call to the desired function of the dll is executed followed by a call to retrieve the error code using pGetErrorFunc. The error code is 0 if no error occurred.

Example

```

module Test;

{structure containing the parameters (just one in this example)}
type
  tstPar = record
    i32Value: Int32;
  end;

private
var
  oNWEvent: toNWEvent;

procedure vMain;
var
  stLFDescr:  tstLFDescr;      {structure containing the function description}
  stLFSetErr: tstLFDescr;     {structure for the error function SetError}
  stLFGetErr: tstLFDescr;     {structure for the error function GetError}
  stErrDescr: tstErrDescr;    {error structure}

```

CallFunc_NW

```

dwRes:      DWord;
stPar:      tstPar;          {structure containing the parameters}

const
      {table to translate error values }
  adwSockMMO: array[0..1*3] of DWord = (1, DWORD_MAX,100,1);

begin

  stLFDescr.pstLibDescr := LoadLib('TestDLL');

  {in case the library could not be loaded}
  if stLFDescr.pstLibDescr = nil then
    Writeln('Error: Could not load library - ', GetErrorMsg(GetError));
    return;
  endif;

  {assign the library to the error function structures}
  stLFSetErr.pstLibDescr := stLFDescr.pstLibDescr;
  stLFGetErr.pstLibDescr := stLFDescr.pstLibDescr;

  {retrieve the address of the TESTDLL_i32GetValue function}
  if not GetLibFunction(stLFDescr, stLFDescr.pstLibDescr,
    'TESTDLL_i32GetValue') then
    Writeln("Function i32GetValue not found",
      GetErrorMsg(GetError));
    return;
  endif;

  {retrieve the address of the TESTDLL_dwGetError function}
  if not GetLibFunction(stLFGetErr, stLFGetErr.pstLibDescr,
    'TESTDLL_dwGetError') then
    Writeln("Function dwGetError not found ", GetErrorMsg(GetError));

    return;
  endif;

  {retrieve the address of the TESTDLL_vSetError function}
  if not GetLibFunction(stLFSetErr, stLFSetErr.pstLibDescr,
    'TESTDLL_vSetError') then
    Writeln("Function vSetError not found ", GetErrorMsg(GetError));
    return;
  endif;

  {assign the error functions to the error structures}
  stErrDescr.pSetErrorFunc := stLFSetErr.pFuncAddr;
  stErrDescr.pGetErrorFunc := stLFGetErr.pFuncAddr;

  {assign the adjustment table}
  stErrDescr.padwMMO := tpaDWord(@adwSockMMO);

  {assign the error structure to the function structure}
  SetLibErrDescr(stLFDescr.pstLibDescr, @stErrDescr);

  {call the external function (this calls the error functions automatically)}
  stPar.i32Value := 150;
  CallFunc_NW(oNWEvent, stLFDescr.pFuncAddr, @stErrDescr,
    @stPar, SizeOf(stPar));

```

CallFunc_NW

```
{wait for the result}
if (WaitEvent(oNWEvent) = nil) then
  Writeln(GetErrorMsg(GetError));
  return;
endif;
dwRes := oNWEvent.unVal.dw;
Writeln(dwRes);

if oNWEvent.dwErrNo<>0 then
  SetError(oNWEvent.dwErrNo); {set the error code according to dwErrNo}
endif;
Writeln(GetErrorMsg(GetError)); {print the error message}

UnloadLib(stLFDescr.pstLibDescr);
end;

procedure vDeinit;
begin
  oNWEvent.vDone;           {Deinitialize the NoWait object}
end;

begin
  oNWEvent.poInit;         {initialize the NoWait object}
end.
```

SetLibErrDescr

Declaration

```
procedure SetLibErrDescr(pstLibDescr: tpstLibDescr; pstErrDescr:
tpstErrDescr);
```

Parameter

pstLibDescr.

tpstLibDescr. A pointer to a structure that describes a library. The structure is to be initialized with a call to LoadLib.

pstErrDescr.

tpstErrDescr.

Pointer to a structure that is used to administer the errors of Lib and NoWait functions. You have to initialize the structure with the error handling functions of the dll and a table to translate the error codes (see the example).

The table is used to translate dll error codes to application (resp. POOL) error codes. The table is composed as follows (see also the description of the tstErrDescr structure):

Min: Minimum value of error codes in the dll.

Max: Maximum value of error codes in the dll.

Offset: Added to the value of the dll for adjustment purpose.

Return value

None.

Description

The procedure associates the Lib's and NoWait's error administration structure with the library identified by the pstLibDescr structure.

Remarks

Libraries that do not return errors or that use a parameter to return errors .

In all other cases you have to provide the information on error handling via the stErrDescr parameter. The error code handling has to be processed in conjunction with the external function call, since different POOL tasks and threads can access the same library.

Example

(see also the example to the CallFunc function);

SetLibErrDescr

```
{retrieve the address of the TESTDLL_dwGetError function}
if not GetLibFunction(stLFGetErr, stLFGetErr.pstLibDescr,
                    'TESTDLL_dwGetError') then
    Writeln("Could not find the dwGetError function ", GetLastErrorMsg(GetError));
    return;
endif;

{retrieve the address of the TESTDLL_vSetError function}
if not GetLibFunction(stLFSetErr, stLFSetErr.pstLibDescr,
                    'TESTDLL_vSetError') then
    Writeln("Could not find the vSetError function ", GetLastErrorMsg(GetError));
    return;
endif;

{assign the error function to the error structure}
stErrDescr.pSetErrorFunc := stLFSetErr.pFuncAddr;
stErrDescr.pGetErrorFunc := stLFGetErr.pFuncAddr;

{assign the adjustment table}
stErrDescr.padwMMO := tpaDWord(@adwSockMMO);

{assign the error structure to the function structure}
SetLibErrDescr(stLFDescr.pstLibDescr, @stErrDescr);
```

16 Compiler internal standard functions

This chapter describes functions and procedures that are not part of a library instead they are executed by the compiler. This is reasonable since the compiler itself creates machine code for the virtual machine, thus running faster.

16.1 General functions

Declaration

```
function Abs (x): Type_x;
```

Parameter

x.

A value whose absolute value is to be returned. Use the function with integer and real variables.

Return value

The absolute value of the parameter. The data type is the same as the type of the parameter.

Description

The function returns the absolute value of the passed parameter. The data type of the return value is the same as of the parameter.

Remarks

Passing an invalid data type (e.g. a pointer) is prevented by the compiler.

Example

```
procedure vMain;
var
  i32Test: Int32;

begin
  i32Test := -1;
  Writeln(Abs(i32Test)); // prints: 1
end;
```

Chr**Declaration**

```
function Chr (x): Char;
```

Parameter

x.

Integer variables (Integer, Word etc.)

Return value

Char.

Low byte of the passed parameter as character (Char). In case the parameter is of size 1 byte the return value is the a conversion to Char.

Description

The function converts the low byte of the passed integer parameter into an ANSI character. The high bytes are truncated. Parameters that have a size of one byte are simply casted. The counterpart to Chr is Ord.

Remarks

Real parameters are prevented by the compiler.

Example

```
procedure vMain;
var
  i32Test: Int32;
  wTest:   Word;
  cTest:   Char;
begin
  i32Test := -191;           {-191 = 0xFFFFFFFF41}
  cTest   := Char(i32Test); {convert low byte to Char (Byte(-191) = 65)}
  Writeln(cTest);          {prints: A}
  wTest   := 41025;         {41025 = 0xA041}
  cTest   := Char(wTest);  {convert low byte to char (0x41=65)}
  Writeln(cTest);          {prints: A}
  Writeln(Ord("A"));      {prints: 65}
end.
```

Ord**Declaration**

```
function Ord (x): ?tType;
```

Parameter

x.

Integer variable. Usually you use the function with parameters of type `Char`.

Return value

?tType.

Depends on the data type of the parameter. The ANSI code is returned in case of a `Char` variable as parameter.

Description

The function returns the ANSI code of the passed `Char` variable. Other integer types as parameter (e.g. `Int32`) are returned without modification. The counterpart to `Ord` is `Chr`.

Remarks

None.

Example

```
procedure vMain;
begin
  Writeln(Ord("A")); {prints: 65}
  Writeln(Chr(65)); {prints: A}
  Writeln(Ord("?")); {prints: 63}
  Writeln(Chr(63)); {prints: ?}
  Writeln(Ord(13)); {prints: 13}
end;
```

Dec**Declaration**

```
procedure Dec (var x [:n]);
```

Parameter

x.

As reference. An integer variable that is to be decremented.

n.

Optional. Default value is 1. Integer variable or constant which is subtracted from the parameter x. $((x)-(n)=(x-n), ((x)-(-n)=(x+n))$.

Return value

None.

Description

The procedure decrements an integer variable by n. The variable is decremented by 1 if n is omitted. The variable is incremented if n is negativ.

Remarks

Notice that there are no checks whether an overflow occurs.

Example

```
procedure vMain;
var
  i32Test: Int32;
  wTest:   Word;
  cTest:   Char;
  r32Test: Real32;

begin
  i32Test := -100; {set variable to -100}
  Dec(i32Test,2);  {decrement by 2}
  Writeln(i32Test); {prints: -102}

  Dec(i32Test,-10); {decrement by -10 (-10 -> increment by 10)}
  Writeln(i32Test); {prints: -92}

  cTest := "B";    {set variable to B}
  Dec(cTest);      {decrement by 1}
  Writeln(cTest);  {prints: A}

end;
```

Inc**Declaration**

```
procedure Inc (var x [:n]);
```

Parameter

x.

As reference. An integer variable that is to be incremented.

n.

Optional. Default value is 1. Integer variable or constant which is added to the parameter x. $((x)+(n)=(x+n))$, $((x)+(-n)=(x-n))$.

Return value

None.

Description

The function increments an integer variable by n. The variable is incremented by 1 if n is omitted. The variable is decremented if n is negativ.

Remarks

Notice that there are no checks whether an overflow occurs.

Example

```
procedure vMain;
var
  i32Test: Int32;
  wTest:   Word;
  cTest:   Char;
  r32Test: Real32;
begin
  i32Test := -100; {set variable to -100}
  Inc(i32Test,2);  {increment by 2}
  Writeln(i32Test); {prints: -98}

  Inc(i32Test,-10); {increment by -10 (-10 -> decrement by 10)}
  Writeln(i32Test); {prints: -108}

  cTest := "B";    {set variable to B}
  Inc(cTest);      {increment by 1}
  Writeln(cTest);  {prints: C}
end;
```

Inc

SizeOf**Declaration**

```
function SizeOf (x): ?t31Bit;
```

Parameter

x.

Any variable or data type description whose size in bytes is to be returned.

Return value

?t31Bit entspricht Int32.

Size of the passed data type in bytes.

Description

The function retrieves the size of the passed data type in bytes. You can pass the data type (e.g. Int32) as well as a variable.

Remarks

The function handles simple data types as well as complex data types like arrays, structures, and objects.

Example

```
module Test;
{Strukturdeklaration}
type
  tstPar = record
    i32Value: Int32;
    r64Value: Real64;
  end;

private

procedure vMain;
var
  r32Test: Real32;
  stPar: tstPar;
  oRoot: tpoRoot;
  ai32Test: array[0..3] of Int32;

begin
  Writeln(SizeOf(r32Test)); {prints: 4}
  Writeln(SizeOf(stPar)); {prints: 12}
  Writeln(SizeOf(oRoot)); {prints: 4}
  Writeln(SizeOf(ai32Test)); {prints: 16}
  Writeln(SizeOf(Real32)); {prints: 4}
  Writeln(SizeOf(tstPar)); {prints: 12}
```

SizeOf

```
Writeln(SizeOf(tpoRoot)); {prints: 4}  
end;
```

TypeOf

Declaration

```
function TypeOf (x):tpstType;
```

Parameter

x.

Any variable or data type.

Return value

tpstType.

Pointer to a structure that contains type information to the passed parameter.

Description

The function returns a pointer to a structure that contains the type information of the passed parameter.

Remarks

None.

Example

```
procedure vMain;
var
  r32Test: Real32;

  pstTest: tpstType;           {pointer to structure of type tpstType}
begin
  pstTest := TypeOf(r32Test);  {retrieve type info of r32Test}
  Writeln(pstTest^.csName);   {prints: Real32}
end;
```

16.2 Dynamic memory management

Declaration

```
function Addr (x): Pointer;
```

Parameter

x.

Any variable of function whose address is to be returned.

Return value

Pointer

Untyped pointer. Address of the passed variable or function.

Description

The function returns the address of a function or variable.

Remarks

You can also use the address operator @ with variables.

See Tutorial part 1 for untyped pointers.

Example

```
module Test;

private

procedure test;
begin
  Writeln("Test");
end;

procedure vMain;
var
  i32Test:  Int32;
  pi32Test: ^Int32;           {pointer to integer}

begin
  i32Test  := 10;             {set variable to 10}
  pi32Test := Addr(i32Test); {assign the variable to a pointer}
  pi32Test^ := 12;           {overwrite the value}
  Writeln(i32Test);          {prints: 12}
```

Addr

```
  Writeln(Addr(test));      {print the address of the function}
end;

begin
end.
```

New**Declaration**

```
procedure New (var p: tpVar);
```

Parameter

p.

tpVar as reference. All pointer types are allowed. p holds the address of the allocated memory if successful or `nil` otherwise.

Return value

None.

Description

The procedure allocates a memory block, initializes the memory with 0 and returns a pointer to it. The allocated size depends on the type of the parameter. `nil` is returned if there is not enough free memory available to allocate the requested block.

Remarks

All memory blocks allocated using `New` have to be freed using `Dispose`. For detailed description on how to use dynamic allocate memory see the Tutorial part 1.

Example

```
procedure vMain;
var
  pi32Test: ^Int32;      {pointer of type Int32}

begin
  New(pi32Test);        {allocate memory}
  pi32Test^ := 100;     {assign a value}
  Writeln(pi32Test^);  {print the value}
  Dispose(pi32Test);   {free the memory}
end;
```

New**Declaration**

```
procedure New (var p: Pointer; xoSize: tSize);
```

Parameter

p.

tpVar as reference. p holds the address of the allocated memory if successful or `nil` otherwise.

xoSize.

tSize equals `Int32`. Size in bytes to allocate.

Return value

None.

Description

The procedure allocates a memory block of size `xoSize`, initialized the memory with 0 and returns a pointer to it. `nil` is returned if there is not enough free memory available to allocate the requested memory block.

Remarks

All memory blocks allocated using `New` have to be freed using `Dispose`. For detailed description on how to use dynamic allocate memory see the Tutorial part 1.

Example

```
procedure vMain;
var
  pTest: Pointer;

begin
  New(pTest,2); {allocate memory (2 bytes)}
  Writeln(pTest); {print the address}
  Dispose(pTest); {free the memory}
end;
```

New**Declaration**

```
procedure New (var p: tpObj; polnit: constructor);
```

Parameter

p.

tpObj as reference. Pointer to an object type. p holds the address of the object if successful or nil otherwise.

polnit.

constructor. Constructor of the object to be used to create the object.

Return value

None.

Description

The procedure reserves memory for an object. The constructor to use is passed via polnit.

Remarks

All memory blocks allocated using New have to be freed using Dispose. For detailed description on how to use dynamic allocate memory see the first part of the tutorial, and the meaning of constructors is explained in the second part of the tutorial.

Example

```
module Test

import LibTest           {contains the declaration of the object}

procedure vMain;
var
  poTest:  ptoTest;
begin
  New(poTest,poInit);    {allocate memory for the object}
  Dispose(poTest,vDone); {free th memory }
end;

begin
end.
```

Dispose

Declaration

```
procedure Dispose (var p: Pointer);
```

Parameter

p.

`Pointer` as reference. All pointer types are allowed. p is set to `nil` after the memory has been cleared.

Return value

None.

Description

The procedure frees the memory the passed pointer points to and set the pointer itself to `nil`. Passing the `nil` pointer has no effect.

Remarks

All memory blocks allocated using `New` have to be freed using `Dispose`. For detailed description on how to use dynamic allocate memory see the Tutorial part 1.

Example

```
procedure vMain;
var
  pi32Test: ^Int32;      {pointer of type Int32}

begin
  New(pi32Test);        {allocate memory}
  pi32Test^ := 100;    {assign a value}
  Writeln(pi32Test^);  {print the value}
  Dispose(pi32Test);   {free the memory}
end;
```

Dispose

Declaration

```
procedure Dispose (var p: tpObj; vDone: destructor);
```

Parameter

p.

tpObj as reference. Pointer to an object that is to be deleted.

vDone.

destructor. Destuctor to use to delete the object.

Return value

None.

Description

The procedure calls the destructor of the object to free the object's memory and set the pointer p to `nil`.

Remarks

All memory blocks allocated using `New` have to be freed using `Dispose`. For detailed description on how to use dynamic allocate memory see the first part of the tutorial, and the meaning of destructors is explained in the second part of the tutorial.

Example

```
module Test

import LibTest

procedure vMain;
var
  poTest: ptoTest;
begin
  New(poTest,poInit);    {allocate memory for an object}
  Dispose(poTest,vDone); {free the memory}
end;

begin
end.
```

16.3 Variable parameter lists and untyped parameters

Taken form the pool.pli library.

POOL supports runtime type information i.e. all types and data structure are not only known by the compiler, but can also be retrieved by the application during runtime. This allows to create function like WriteIn in POOL contrary to Pascal. The following chapter describes to to use runtime type information.

16.3.1 Functions and procedures for variable parameter lists

Declaration

```
function VAFirst(var stVAInfo: tstVAInfo; pstVAStart: tpstVArg): Boolean; ifr;
```

Parameter

stVAInfo.

tstVAInfo as reference. A structure that contains information of the parameters of the variable parameter list.

pstVAStart.

tpstVArg. Pointer to the first element of the parameter list. pstVAStart is passed implicitly when a function of procedure with an variable parameter list is called.

Return value

Boolean.

true, if a parameter exists, otherwise false.

Description

The function is used within another function having an variable parameter list, to initialize the stVAInfo structure. This structure contains information of parameter of the variable parameter list (see also the description of the stVAInfo structure). The pointer pstVAStart is passed implicitly when a function of procedure with an variable parameter list is called. It points to the first element of the parameter list.

Remarks

Attention: If an integer constant is passed, it is not necessarily recognized as Int32 type in the following example. See the example to the VANext function that would also work correctly with constants.

The modifier ifr indicates that you can ignore the return value.

Example

(For an example that allows to pass constants see the description of the VANext function.)

<pre>procedure WriteValue(..);</pre>	<pre>{function with a variable parameter list}</pre>
--------------------------------------	--

VAFirst

```

var
  stVAInfo:    tstVAInfo;           {general information structure}
  stUArg:      tstUArg;             {information to the argument}
  pstVA:       tpstVArg;           {pointer to the current argument}
  stVal:       tstVal;             {value of the current element}
  boHelp:      Boolean;

begin
  {initialize stVAInfo}
  if not (VAFirst(stVAInfo,pstVStart)) then
    Writeln("Error - ",GetErrorMsg(GetError));
    return;
  endif;
  pstVA := pstVStart;              {pointer to the current element}
  repeat                            {until all arguments are evaluated}

    GetVA(stUArg,pstVA);           {retrieve the data type of the
                                  argument (in stUArg)}
    if !(GetVAVal(stVal,pstVA)) then {retrieve the value of the argument
                                  (in stVal)}
      Writeln("GetVAVal-Error");   {print error message - tested twice
                                  via vistUArg.pstType^.csName
                                  (in the else branch) but is
                                  only needed once in practice}
    endif;

    {in case a Int32 data type is passed - works only with variables}
    if stUArg.pstType^.csName = "Int32" then
      Writeln(stVal.i32);          {print the i32 value}
    {in case a Real64 data type is passed - works only with variables}
    elseif stUArg.pstType^.csName = "Real64" then
      Writeln(stVal.r64);          {print the r64 value}

    {in case no valid data type is passed - see also GetVAVal above,
     where the error is also evaluated }
    else
      Writeln("Not an evaluated parameter.");
    endif;
    pstVA := PVANext(pstVA);       {get pointer to the next element}

  until pstVA = nil ;              {exit loop if no additional element
                                  is available}
  GetVA(stUArg,pstVA);           {free stUArg}
end;

{main program}
procedure vMain;
var
  i32Count: Int32;
  r64Val:   Real64;
  wTest:    Word;
  ai32Test: array [0..2] of Int32;

begin
  r64Val := 2.3;
  for i32Count := 0 to 2 do
    ai32Test[i32Count] := i32Count;
  end;
end;

```

VAFirst

```
endfor;

{calling the function passing differen types of parameters }
WriteValue(r64Val,ai32Test[0],3,,wTest,ai32Test[1],
           ai32Test[2],r64Val);
{prints:                                     }
{      2.3                                   }
{      0                                     }
{      Not an evaluated parameter           }
{      GetVAVal error                       }
{      Not an evaluated parameter           }
{      Not an evaluated parameter           }
{      1                                     }
{      2                                     }
{      2.3                                   }

end;
```

PVANext**Declaration**

```
function PVANext (pstVA: tpstVArg): tpstVArg;
```

Parameter

pstVA.

tpstVArg. Pointer to the current element of the parameter list. Pass pstVAStart for the first element of the list. Pass the result of the previous call as parameter to process the complete parameter list.

Return value

tpstVArg.

Pointer to the next element of the parameter list or `nil` if on other element is available.

Description

The function returns a pointer to the next element of a variable parameter list or `nil` if on other element is available.

The pointer to the first element of a variable parameter list is passed (pstVAStart). This pointer is passed implicitly to functions that have a variable parameter list. Use a loop to process all parameter contained in a variable parameter list and test for `nil` to exit the loop. Pass the pointer to the current element of the parameter list which is the result of the previous call to PVANext.

Remarks

A pointer to the value is passed instead of the value for variables (`nOFVarParamMask`) or typed constants (`nOFConstParamMask + nOFConstParamMask`) like is done with `tstUArg`.

Pass the type `tpDummyArg` with the value `nil` for optional parameters.

Attention:

Only the current used length is available. Use type casts to access other types than the predefined base types!

Dereference pointers are always passed with `nOFVarParamMask`, since POOL does not have pointers to constants. A function has to be implemented that tests and returns the addressed segment (CODE, CONST, IVAR, VAR, STACK or HEAP)!

The statically known type of the object variable is passed differently than with `TypeOf`. You can access the actual type of the object variable via `unData.pstVOT` (of course only if the object is initialized).

PVANext

Only the current list is processed if no special treatment is provided. Embedded parameter lists and so on has to be handled by the function itself.

Example

(see VAFirst)

GetVA**Declaration**

```
procedure GetVA (var stUA: tstUArg; pstVA: tpstVArg);
```

Parameter

stUA.

tstUArg as reference. Structure containing the type description, the options, and a pointer to the value of the current parameter.

pstVA.

tpstVArg. Pointer to the current element of the parameter list. Pass pstVAStart for the first element of the list. For all additional arguments use PVANext to get pstVA for the current parameter (see the example to PVANext).

Return value

None.

Description

Determine the data type of an argument. A structure of type `tstUArg` is passed as reference to the procedure. The structure contains the type information to the current element after the procedure call (see also the `tstUArg` structure). The pointer `pstVA` is used to point to the argument in the parameter list that is to be evaluated. The first element is reference via `pstVAStart`. The pointer to all next elements has to be retrieved using the `PVANext` function. The structure `stUA` is deleted if `pstVA` is `nil`.

Remarks

Differently than with `pstVA` itself in `stUa` a pointer to the data is always returned even with `val` parameters and the parameter itself is returned for untyped `var` parameters instead of `tstUArg`. Possibly contained lists (`tpstVArg`) can not be handled contrary to `VANext`, i.e. they have to be handled in a superior function (e.g. with recursive calls).

Example

(see `VAFirst`)

GetVAVal

Declaration

```
function GetVAVal (var stVal: tstVal; pstVArg: tpstVArg): Boolean;
```

Parameter

stVal.

tstVal as reference. Structure containing the data and radix of the parameter (stVal.bRadix=0 (real) or 10 (int or enum)).

stVal.enBType (=nenBT7Bit..nenBTdWord,nenBTReal32,nenBTReal64) contains the base type of the data resp. nenBTNNone in case of an error.

The value is contained according to the data type of the parameter in stVal.i32, stVal.dw or stVal.r64

pstVArg.

tpstVArg. Pointer to the current element of the parameter list. Pass pstVASTart for the first element of the list. For all additional arguments use PVANext to get pstVA for the current parameter (see the example to PVANext).

Return value

Boolean.

false in case of an invalid data type and stVal.enBType is set to nenBTNNone. true is returned otherwise.

Description

Retrieve the value of an argument. The function returns an integer, enum or real value via the stVal structure.

Additional you can determine the used base via bRadix (Radix (stVal.bRadix=0 (real) or 10 (int or enum)). The base type of the data can be retrieved via stVal.enBType. stVal.enBType is nenBTNNone in case of an error.

The value of the current data type is contained in stVal.i32, stVal.dw or stVal.r64 according to its type.

Additional to the structure you can use the return value to detect whether a valid data type was passed.

Remarks

Attention: Only the parameter itself is processed directly. In case that the parameter contains a list (e.g. when passing pstVASTart to other functions that have a variable parameter list) they have to be processed separately using VAFirst resp. UAFirst and VANext.

GetVAVal**Example**

(see VAFirst)

VANext**Declaration**

```
function VANext(var stVAInfo: tstVAInfo): Boolean; ifr;
```

Parameter

stVAInfo.

tstVAInfo as reference. Structure containing information to the parameter of the variable parameter list.

Return value

Boolean.

true, in case that additional parameters are available or false otherwise.

Description

The function returns the next parameter of the parameter list via a stVAInfo structure.

Remarks

Untyped parameters, embedded parameter lists and maximum one tstVAInfo structure at the end are handled automatically. The function aborts with an error message in case the maximum nesting level is exceeded (see ?nVAMaxNesting) or a tstVAInfo structure is missing at the end. stVAInfo is deleted if no parameters are available any more, i.e. you can also test pstType for nil or stVa.pvData for nil to detect an abortion criterion.

The following constraints exist when passing a tstVAInfo structure as untyped parameter or as part of a parameter list:

The current parameter to process has to be contained (i.e. the caller possibly has to call VANext previously).

The structure has to be the last parameter. The called function is not allowed to pass the list directly or indirectly with additional parameter.

The modifier ifr indicates that you can ignore the return value.

Example

(Works also when constanst are passed contrary to the example presented with the VAFirst function because of the modified evaluation of the parameters)

procedure WriteValue(..);	{function with a variable parameter list}
var	
stVAInfo: tstVAInfo;	{general information structure}

VANext

```

    stUArg:      tstUArg;           {information to the argument}
    pstVA:       tpstVArg;         {pointer to the current argument}
    stVal:       tstVal;          {value of the current element}
begin
    {Initialisation of stVAInfo}
    if not (VAFirst(stVAInfo,pstVAStart)) then
        Writeln("Error - ",GetErrorMsg(GetError));
        return;
    endif;

    repeat                               {until all arguments are processed}
        {get int, byte or real value from stVAInfo. Result stored is in stVal}
        if GetUVal(stVal,stVAInfo.stVA) then
            if stVal.bRadix=0 then        {in case of a real value }
                Writeln(stVal.r64);

            elseif stVal.enBType = nenBTDWord then {byte value}
                Writeln(stVal.dw);

            else                           {Int32 value}
                Writeln(stVal.i32);

            endif;
        {in case none of the previous data types was passed}
        elseif stVAInfo.stVA.pstType <> TypeOf(tpDummyArg) then

            {determine data type}
            case stVAInfo.stVA.pstType^.enTypeClass of

                nenTCBoolean:              {type boolean}
                begin
                    if tpBool(stVAInfo.stVA.pvData)^ then
                        Writeln("True");
                    else
                        Writeln("False");
                    endif;
                end;

                nenTCCharString:           {type string}
                begin
                    Writeln(tpCharStr(stVAInfo.stVA.pvData)^);
                end;

                else                         {other data type}
                Writeln("Illegal parameter");
                return;

            endcase;
        endif;
        until not VANext(stVAInfo);       {exit loop if there is no additional
                                          element}
    end;

    {main program}
    procedure vMain;

```

VANext

```
var
  i32Count: Int32;
  r64Val:   Real64;
  wTest:   Word;
  i32Test: Int32;
  cTest:   Char;
begin
  r64Val := 2.3;
  i32Test := -2;
  wTest := 34;
  cTest := "A";

  {calling the function passing different parameters }
  WriteValue(r64Val,i32Test,3,,wTest,r64Val,"Teststring",true,"A");
  {prints:
  {
    {      2.3
    {      -2
    {      3
    {      34
    {      2.3
    {      Teststring
    {      True
    {      Illegal parameter
  }
end;
```

GetVANext

Declaration

```
function GetVANext (var stUA: tstUArg; var pstVA: tpstVArg): Boolean;
```

Parameter

stUA.

tstUArg as reference. A structure containing the type description, the options and a pointer to the value of the current parameter.

pstVA.

tpstVArg as reference. Pointer to the current element of the parameter list. Pass pstVStart for the first element of the parameter list. This parameter is passed implicitly to each function that has a variable parameter list and can be used directly.

Return value

true if a parameter is passed otherwise false.

Description

The function GetVANext retrieves some pointers to the next element of the parameter list. A pointer to the type description, a pointer to the value, and the options to the current parameter of the stUS structure (see the description of the tstUS structure). A pointer to the current parameter is passed to the function. This pointer is set to the next parameter provided there is one. In this case true is returned and false otherwise.

Remarks

None.

Example

(See VAFirst).

16.3.2 Functions and procedures for untyped parameters

Declaration

```
function UAFirst(var stVInfo: tstVInfo; var stUArg: tstUArg): Boolean; ifr;
```

Parameter

stVInfo.

tstVInfo as reference. Structure that receives the information of the untyped parameter.

stUArg.

tstUArg as reference. Untyped parameter, that is passed to the function.

Return value

Boolean.

true if a parameter is passed otherwise false.

Description

The function initializes stVInfo using the passed untyped parameter. The structure is the used to get additional information (see GetUAVal).

Remarks

The modifier `ifr` indicates that you can ignore the return value.

Example

(see VANext).

GetUAVal**Declaration**

```
function GetUAVal (var stVal: tstVal; stUArg: tstUArg): Boolean;
```

Parameter

stVAInfo.

tstVAInfo as reference. Structure that receives the information of the untyped parameter.

stUArg.

tstUArg as reference. The Untyped parameter consists of a structure that contains a type description, options and a pointer to the current value.

Return value

Boolean.

true in case of an valid data type. Otherwise false is returned and stVal.enBType is set to nenBTNone.

Description

The function returns the value (integer, real or DWord) via the stVal structure. The return value is true if the data type is one of previous mentioned types and false otherwise. Passing constanst is prevented by the compiler.

Remarks

stVal.enBType = nenBT7Bit..nenBTDWord, nenBTReal32, nenBTReal64
stVal.bRadix=0 (real) or 10 (int or enum). Value in stVal.i32, stVal.dw or stVal.r64.

Attention:

Only the parameter itself is processed directly. In case that the parameter contains a list (e.g. when passing pstVAStart to other functions that have a variable parameter list) they have to be processed separatly using VAFirst resp. UAFirst and VANext.

Example

(see also UAFirst)

<pre>procedure WriteValue(var test); var stVal: tstVal;</pre>	<pre>{procedure with an untyped parameter} {value of the current element}</pre>
---	---

GetUAVal

```
begin
  if GetUAVal (stVal,test) then      {if data type is valid}
    if test.pstType^.csName = "Int32" then
      Writeln(stVal.i32);           {print i32 value}

      {if the type passed is of type Real64}
    elseif test.pstType^.csName = "Real64" then
      Writeln(stVal.r64);           {print r64 value}
    else
      Writeln(stVal.dw);           {print dw value}
    endif;

    {in case an invalid parameter is passed}
  else
    Writeln("Invalid parameter.");
  endif;
end;

{main program}
procedure vMain;
var
  r64Test:  Real64;
  i32Test:  Int32;
  dwTest:   DWord;
  csTest:   String;

begin
  r64Test := 2.3;
  i32Test := -2;
  dwTest  := 36;
  csTest  := "Teststring";

  {call the function using different parameters}
  WriteValue(r64Test);           {prints: 2.3           }
  WriteValue(i32Test);           {prints: -2         }
  WriteValue(dwTest);            {prints: 36        }
  WriteValue(csTest);            {prints: Invalid ...}
end;
```

17 Miscellaneous functions

17.1 Val function

Declaration

```
procedure AdaptVal(var stVal: tStVal; bBTF: Byte);
```

Parameter

stVal.

tStVal as reference. A structure that contains the value that is to be converted according to the bBTF flag (see also the description of the tStVal structure).

bBTF.

Byte. A flag that specifies the type that the value should be converted to.

Possible flag values are:

nBTFReducedInt = 0x10 Integer with limited range.

nBTFUnsignedInt = 0x20 Integer with unsigned range.

nBTFSignedInt = 0x40 Integer with signed range.

nBTFReal = 0x80 Real.

Return value

None.

Description

The procedure converts the value in stVal into the type that is provided via the bBTF flag. The function result is stVal.enBType = nenBTNNone if the bBTF is invalid (or if stVal.enBType is already nenBTNNone) resp. a type, that is allowed, but the value was set to an allowed value (e.g. in case the range was exceeded).

Remarks

None.

Example

```
procedure vMain;
```

AdaptVal

```
var
  stVal: tstVal;

begin
  stVal.enBType := nenBTInt32;
  stVal.bRadix  := 10;
  stVal.i32     := -23;
  Writeln(stVal);           {stVal.i32 := -23, stVal.r64 := 0}
  AdaptVal(stVal,nBTFReal); {convert to Real }
  Writeln(stVal);           {stVal.i32 := 0, stVal.r64 := -23}
end;
```

17.2 Memory functions

Declaration

```
function DataSize (pstType: tpstType; pvData: Pointer): ?t31Bit;
```

Parameter

`pvData`.

`tpstType`. Pointer to a structure that contains the information of the type (see also `tpstType`). You can pass the variable to the function `TypeOf` that returns a `tpstType` structure describing the variable that was passed.

`pvData`.

`Pointer`. A pointer to a block whose size is to be returned.

Return value

`?t31Bit`.

Size of the data block.

Possible values:

0:

`pstType = nil` or `pvData = nil` or `pstType^.xoSize=0`.

`HeapBlockSize(pvData)`:

`pvData` points to a valid POOL heap block and `HeapBlockSize(pvData) < pstType^.xoSize`.

`nMaxOASize`:

Open array of an unknown size.

`pstType^.xoSize`: otherwise.

DataSize**Description**

The function returns the size that is used up by the block pvData points to. The type of the pointer is passed using the a pointer to the structure pstType. You can pass the structure using the TypeOf function (see the example).

Remarks

None.

Example

```
procedure vMain;
var
  pi32Val:      ^Int32;
  i32DataSize: Int32;

begin
  New(pi32Val);           {allocate memory}
  i32DataSize := DataSize(.TypeOf(Int32),pi32Val); {get size of memory}
  Writeln(i32DataSize);  {prints: 4}
  Dispose(pi32Val);      {free memory}
  i32DataSize := DataSize(.TypeOf(Int32),pi32Val); {get size of memory}
  Writeln(i32DataSize);  {prints: 0}
end;
```

HeapBlockSize

Declaration

```
function HeapBlockSize (p: Pointer): ?t31Bit;
```

Parameter

p.

Pointer. Pointer to the heap block.

Return value

?t31Bit.

Size of the heap block. The function returns 0 if the pointer does not point to a heap block.

Description

The function returns the size of the heap block. The function returns 0 if the pointer does not point to a valid heap block.

Remarks

None.

Example

```
procedure vMain;
var
  pi32Val:      ^Int32;
  i32DataSize: Int32;

begin
  New(pi32Val);           {allocate memory}
  i32DataSize := HeapBlockSize(pi32Val); {get the number of allocated bytes}
  Writeln(i32DataSize);  {prints: 4}
  Dispose(pi32Val);      {free the memory}
  i32DataSize := HeapBlockSize(pi32Val); {get the number of occupied bytes}
  Writeln(i32DataSize);  {prints: 0}
end;
```

MemCmp**Declaration**

```
function MemCmp (var x1,x2; xoCount: tSize): Int16;
```

Parameter

x1.

Any type as reference.

x2.

Any type as reference.

xoCount.

tSize equals Int32. Number of bytes to compare. The number of bytes of the smaller variable is compared if xoCount < 0.

Return value

Int16.

Possible results:

> 0 if x1 > x2 or (x1 = nil and x2 <> nil).

= 0 if x1 = x2 or xoCount=0.

< 0 if x1 < x2 or (x1 <> nil and x2 = nil)

Description

The function compares xoCount bytes of two memory blocks.

Remarks

The size of the object is retrieved automatically, based on the real object type when xoCount < 0 instead of using the current (static) type.

xoCount is limited to the size of the smaller heap block if x1 or x2 are heap blocks.

When comparing non empty (Byte)Strings, the result is 0 (equal) only when xoCount=0 or @x1 = @x2. Different pointers are stored in the buffer even if the strings have the same content since POOL uses dynamic strings.

Example

```
procedure vMain;
var
```

MemCmp

```
    i32Val1: Int32;  
    i32Val2: Int32;  
begin  
    i32Val1 := 100;  
    i32Val2 := 99;  
    Writeln(MemCmp (i32Val1,i32Val2,-1)); {prints: 1}  
end;
```

MemMove**Declaration**

```
procedure MemMove (var xSource,xDest; xoCount: tSize);
```

Parameter

xSource.

Any data type as reference.

xDest.

Any data type.

xoCount.

tSize equals Int32. Number of bytes to copy. SizeOf(xDest) bytes are copied if xoCount < 0.

Return value

None.

Description

The procedure copies xoCount bytes from xSource to xDest. SizeOf(xDest) bytes are copied if xoCount < 0.

Remarks

If xDest is a structure containing strings then the strings are copied. This is only possible when xSource and xDest have the same structure.

The size of the object is retrieved automatically, based on the real object type when xoCount < 0 instead of using the current (static) type.

xoCount is limited to the size of the smaller heap block if xSource or xDeast are heap blocks.

Example

```
procedure vMain;
var
  i32Val1:  Int32;
  i32Buffer: Int32;

begin
  i32Val1 := 100;
  MemMove(i32Val1,i32Buffer,-1);
```

MemMove

```
Writeln(i32Val1,"",i32Buffer); {prints: 100,100}
end;
```

MemSet**Declaration**

```
procedure MemSet (var xBuf; bVal: Byte; xoCount: tSize);
```

Parameter

xBuf.

Any data type as reference. A memory block, that is to be set to the data provided via bVal.

bVal.

Byte. Data to be written into memory.

xoCount.

tSize equals Int32. Number of bytes to write. SizeOf(xBuf) bytes are set to bVal if xoCount < 0.

Return value

None.

Description

The procedure sets xoCount bytes of xBuf to bVal. SizeOf(xBuf) bytes are set to bVal if xoCount < 0.

Remarks

The size of the object is retrieved automatically, based on the real object type when xoCount < 0 instead of using the current (static) type.

xoCount is limited to the size of the heap block if xBuf is a heap block.

Example

```
procedure vMain;
var
  wBuffer: Word;
begin
  MemSet (wBuffer, 255, 1);
  Writeln (wBuffer);      {prints: 255}
  MemSet (wBuffer, 255, 2);
  Writeln (wBuffer);      {prints: 65535}
  MemSet (wBuffer, 255, -1);
  Writeln (wBuffer);      {prints: 65535}
```

MemSet

```
end;
```

GetKinship

17.3 Relationship of objects

Declaration

```
function GetKinship(pstType1, pstType2: tpstType): tenKinship;
```

Parameter

pstType1.

tpstType. Record or object type whose relationship to pstType2 is to be checked.

pstType2.

tpstType. Record or object type whose relationship to pstType1 is to be checked

Return value

tenKinship.

Mögliche Werte:

nenKSNone: Wrong types (no record or object resp. different type classes)

nenKSNoInit: At least on type =nil (e.g. object is not initialized).

nenKSStrange: Type1 does not know type2 (i.e. type 1 and type2 do not have a relationship or type 1 is an ancestor of type2).

nenKSDescendant: Typ1 is descendant of Typ2.

nenKSEqual: Typ1 = Typ2.

Description

The function checks the relationship of records and objects. The possible result can be taken from the return value.

Remarks

None.

Example

```
module Test;
```

GetKinship

```
private
var
  oRoot:    toRoot;
  oNWEvent: toNWEvent;

procedure vMain;
var
  enKinship: tenKinship;

begin
  {check relationship}
  enKinship := GetKinship(.TypeOf(oNWEvent), TypeOf(oRoot));
  Writeln(enKinship);           {prints: nenKSDescendant}
end;

procedure vDeinit;
begin
  oRoot.vDone;
  oNWEvent.vDone;
end;

begin
  oRoot.poInit;
  oNWEvent.poInit;
end.
```

HexDump

17.4 Converting into hexadecimal format

Declaration

```
procedure HexDump (var fiOut: tFile; dwAddr: DWord; bAddrLen: Byte; var xBuf;  
xoCount: tSize; bIndent, bBPL, bASCII: Byte);
```

Parameter

fiOut.

tFile as reference. Handle to the output file (use StdOut to print to the console).

dwAddr.

DWord. An address that is to be put out (possibly) before the data

bAddrLen.

Byte. Number of bytes to put out the address or 0 if no address is to be put out.

xBuf.

Any type as reference. A variable whose content is to be put out.

xoCount.

tSize equals Int32. Number of data bytes to put out. SizeOf(xBuf) bytes are put out if xoCount < 0 and for objects the automatic calculated number of bytes for the actual type of the object.

bIndent.

Byte. Number of spaced used to indent the output.

bBPL.

Byte. Number of bytes to put out per line or 0 if all bytes are put out without a line feed character.

bASCII:

Format to use: 0 = no ASCII, 7=ASCII 7-Bit (0x20..0x7E), 8=ASCII 8-Bit (0x20..0xFF).

HexDump

Return value

None.

Description

The procedure puts out any variable in hexadecimal format. You can specify the output format via several parameters. The output is done either to a file or to the standard output.

Remarks

You have to specify the index and the length to put out Char or ByteString, e.g. HexDump(StdOut,0,0,csStr[0],Length(csStr),0,0,7).

SizeOf(xBuf) bytes are put out if xoCount < 0 and for objects the automatic calculated number of bytes for the actual type of the object.

If xBuf is a heap block then xoCount is limited to the heap block size.

Example

```
procedure vMain;
var
  csStr: String;

begin
  csStr := "Teststring";
  HexDump(StdOut,0,0,csStr[0],Length(csStr),0,1,8);
end;
```

17.5 Random number function

Declaration

```
function Random (i16Lim: Int16): ?t15Bit;
```

Parameter

i16Lim.

Int16. Specifies the range of the resulting value.

Possible values are:

0 <= random number < i16Lim, if i16Lim > 0

0 <= random number < 32767, if i16Lim <= 0

Return value

?t15Bit.

Random number within the passed range.

Description

The function return a random number whose range is between 0 and i16Lim if i16Lim > 0 and between 0 and 32767 if i16Lim <= 0.

Remarks

The function Randomize (see next page) has to be called if a new thread is started, since the thread would return the same sequence of random numbers without this explicit call.

Different tasks and thread do not influence each other.

Example

```
procedure vMain;
begin
  repeat
    Writeln(Random(100));      {prints:0 <= random number < 100}
    Writeln(Random(0));      {prints:0 <= Random number < 32768}
    Wait(1000);              {wait for one second}
  until KeyPressed <> 0
end;
```

Randomize

Declaration

```
procedure Randomize;
```

Parameter

None.

Return value

None.

Description

The procedure initializes the random number generator.

Remarks

The procedure is called automatically during the start of every POOL tasks. The procedure is not called automatically when a thread starts, thus the thread inherits the current state (i.e. the thread generates the same random numbers if Randomize was not called explicitly)

Example

```
{call once in a thread if you want to use random numbers}  
Randomize;
```

17.6 Conversion using conversation vectors

Declaration

```
procedure VecConv(var xBuf; bsConvVec: ByteString);  
procedure VecConvI(var xBuf; bsConvVec: ByteString);
```

Parameter

xBuf.

ByteString as reference. Data that is to be converted into another format.

bsConvVec.

ByteString. Conversion vector.

Return value

None.

Description

The functions convert a buffer using a conversion vector.

The function is used in the normalization module to convert data that is communicated from a controller to the host into host byte order (and the other way back). Data with identical binary representation (except for the byte order) can be converted using vectors describing the different types e.g. a device using a Motorola processor (big endian) communication with an X86 PC (little endian). Little endian and big endian differ in the way the single data bytes are arranged in memory. Starting at the variable address big endian orders the bytes starting at the high byte whereas little endian starts with the low byte.

The vector for conversion from a little endian host to a big endian controller is e.g:
(Conversion vectors are defined in the NORM library)

And the command

```
VecConv (tpaByte(pvData)^, paabsVector^[NORM_nenBOTWord16,enDirection]);
```

converts the 16 bit integer value in pvData.

The function VecConvI reconverts a previously converted data type using VecConv to its original value.

VecConv**Remarks**

For a description of the data types etc. see the description of the norm.pli library.

The function has no effect if length=0, length > 256 or Addr(xBuf) = nil.

Remarks to ConvVec

The length of bsConvVec determines the number of converted bytes.

In case that the same value is contained in bsConvVec, the resulting buffer gets the same value assigned to two different locations in accordance to the vector.

The buffer is not modified if the vector contains the values 0,1,2,.. (in that particular order). The conversion returns the conversion vector if the buffer contains the value 0, 1, 2,

Remarks to ConvVecI

The biggest value in bsConvVec determines the number of converted bytes.

In case that the same value is contained in bsConvVec, the resulting buffer contains only the value that was written last to this location in the buffer.

Positions that are not written to are deleted (e.g. because bsConvVec has the same values more often than once or if value are greater or equal then the length).

VecConvI 'undos' the conversion made by VecConv if the vector contains the numbers 0 .. Length -1

Example

See the description of the library norm.pli in the documentation of the advanced libraries.

18 Constants and masks from pool.pli

Taken from the pool.pli library.

The description of the masks and constants are taken from the pool.pli library with almost no extensions.

18.1 Standard constants (internal to the compiler)

```
nil      = Pointer(0);  
nil is the nil pointer.  
  
pi       = 2*ArcCos(0);  
The number pi used for calculations.  
  
false = (1=0);  
false (used for boolean values).  
  
true  = (1=1);  
true (used for boolean values).
```

18.2 System and Commander constants

```
stPIInfo: tstPIInfo = ();  
Version information to PI and operating system (fields are set by PI).  
  
stTZInfo: tstTZInfo =  
acsNames:('', '');i16MinutesWest:0;i8IsDst:0;bDummy:0;dwTime:0);  
Information to the current time zone. The information is retrieved during  
startup, thus dwTime is the start time of the PI. This time remains valid  
during runtime since there are no PI functions to change the time zone, only  
i8IsDst is only valid for dwTime (DST state (e.g. of the current time) is  
delivered by MakeDateTime).
```

18.3 General constants and masks

```
nwTimeoutInfinite = 0xFFFF;  
Infinite timeout for WaitKey.
```

```
unLastKeyVal: tunKeyVal;
Last value of KeyPressed etc.

nMaxOASize = 0x70000000;
Adequate maximum size for 'open' arrays (i.e. arrays of variable size, which
have the size that is actually needed, usually stored on the heap).
Remarks:
All arrays (array[...] of Field), having xOSize >= nMaxOASize-SizeOf(Field),
are handled internal as open arrays.
nMaxOASize < SIZE_MAX, allows to use open array within records, e.g. header
plus open array for the data in communication buffers.
Diverse function take the real size (on the heap) into account if possible.
Attention: If used with records it is possible that the functions can not
determine the real size and therefore use SizeOf(field), i.e. just one
element, or the abort.

BYTE_MAX      = 0xFF;
Maximum value of the range of a Byte variable.

WORD_MAX      = 0xFFFF;
Maximum value of the range of a Word variable.

DWORD_MAX     = 0xFFFFFFFF;
Maximum value of the range of a DWord variable.

INT8_MIN      = -0x7F-1;
Minimum value of the range of an Int8 variable.

INT8_MAX      = +0x7F;
Maximum value of the range of an Int8 variable.

INT16_MIN     = -0x7FFF-1;
Minimum value of the range of an Int16 variable.

INT16_MAX     = +0x7FFF;
Maximum value of the range of an Int16 variable.

INT32_MIN     = -0x7FFFFFFF-1;
Minimum value of the range of an Int32 variable.
INT32_MAX     = +0x7FFFFFFF;
Maximum value of the range of an Int32 variable.

SIZE_MAX      = +0x7FFFFFFF;
Maximum value of the range of a tSize variable.

Result types of TypeOf.

nMaxIDLen = 63;
Signifikant characters of all designators.

acEmptyHStr: array[0..0] of Char = ('\0');
Empty HStr for LibCall etc.

?sTempStr: _tstCharString = (pacCont:nil; dwFlgLen:$80000000);
sTempStr: Template for the compiler (for (Byte)String functions)
```

```
bsEmptyBStr: ByteString = ();  
Empty ByteString for comparison etc. (especially since there are no immediate  
ByteStrings).  
  
hOwnTHandle: tTHandle = nil;  
Own POOL task handle.
```

18.4 Constants for variable parameter lists and untyped parameters

Masks and values for tstOptions.bFlags

```
nOptTypeMask = $03;
Two bits for the type of each option.

abOTShift: array[0..2] of Byte = (0,2,4);
Shift faktor for accessing the OptType bits.

nOTNone = 0;
Option not stated.

nOTInt8 = 1;
Option stated as Int8.

nOTByte = 2;
Option stated as Byte.

nOTChar = 3;
Option stated as Char.

nOFVarParamMask = $40;
Var parameter, i.e. address is passed.
Attention: 1. It is no "real" Var parameter if nOFConstParamMask is set. In
this case it is not allowed to modify the parameter. Constant Char strings
are passed that way for instance.
2. A dereferenced pointer can point into the const segment, without having
set the nOFConstParamMask!

nOFConstParamMask = $80;
constant.

Dummy options

stDummyValOpt: tstOptions = (bFlags:0; ai8Val:(0,0,0));
Dummy option for value parameters (value is passed).

stDummyVarOpt: tstOptions = (bFlags:nOFVarParamMask; ai8Val:(0,0,0));
Dummy option for Var parameters (address is passed).

stDummyTCOpt:  tstOptions = (bFlags:nOFConstParamMask or nOFVarParamMask;
                           ai8Val:(0,0,0));
Dummy option for typed constants (address within the const segment is
passed).

?nVAMaxNesting = 6;
```

Maximum nesting level for VarArg. Simply passing pstVStart is not counted, since an address has to be stored (for processing the parameter) only if additional parameters are attached.

18.5 File constants and masks

```
nInvFHandle = 0;  
Invalid file handle.
```

```
nFFPresNLMask = 0x02;  
Sustain end of line character when using the TRead, TReadln functions.
```

```
nFFLFMask      = 0x10;  
LF ("\n") (also CR+LF) was removed from the end.
```

```
nFFCRLFMask   = 0x20;  
CR+LF ("\r\n") was remove from the end.
```

Remarks:

If a file is opened as text file on the Windows operating system, the standard C functions convert CR+LF to LF, thus nFFCRLFMask is not set.

If a file is opened as text file on the Windows operating system, the standard C functions truncate the Ctrl-Z ('\x1A') that indicates the EOF. The same file opened as binary file or on a Unix system would return the Ctrl-Z character as data byte.

18.6 Control character constants

```
ncSOH  = '\x01';  
^A  
  
ncSTX  = '\x02';  
^B      (MAP27, RVI-Hex)  
  
ncETX  = '\x03';  
^C      (MAP27)  
  
ncEOT  = '\x04';  
^D  
  
ncENQ  = '\x05';  
^E  
  
ncACK  = '\x06';  
^F  
  
ncBEL  = '\x07';  
^G      '\a' (alert)  
  
ncBS   = '\x08';  
^H      '\b'  
  
ncHT   = '\x09';  
^I      '\t' (tab)  
  
ncLF   = '\x0A';  
^J      '\n'  
  
ncVT   = '\x0B';  
^K      '\v'  
  
ncFF   = '\x0C';  
^L      '\f'  
  
ncCR   = '\x0D';  
^M      '\r'  
  
ncSO   = '\x0E';  
^N  
  
ncSI   = '\x0F';  
^O  
  
ncDLE  = '\x10';  
^P      (MAP27)  
  
ncDC1  = '\x11';  
^Q      (XON)
```

```
ncDC2   = '\x12';  
^R  
  
ncDC3   = '\x13';  
^S      (XOFF)  
  
ncDC4   = '\x14';  
^T  
  
ncNAK   = '\x15';  
^U  
  
ncSYN   = '\x16';  
^V      (MAP27)  
  
ncETB   = '\x17';  
^W  
  
ncCAN   = '\x18';  
^X  
  
ncEM    = '\x19';  
^Y  
  
ncSUB   = '\x1A';  
^Z      used as END OF FILE in CPM/DOS text files  
  
ncESC   = '\x1B';  
^[  
  
ncFS    = '\x1C';  
^\  
  
ncGS    = '\x1D';  
^]  
  
ncRS    = '\x1E';  
^^  
  
ncUS    = '\x1F';  
^_  
  
ncDEL   = '\x7F'
```

18.7 Constants for library functions

```
NInvHandle = nil;
Invalid handle (deleted memory has to be nInvHandle!)
```

18.8 Bitmasks for tstType.bFlags

```
nTFStrsContMask = $80;
the data type contains strings. The upper bits are also valid for basic data
types (always = 0).
```

18.9 Bitmasks for tstField.wFlags

```
nFFPrivateMask = $0020;
Private component.

nFFVariantMask = $0040;
The field is the first field of a Variant

nBTFSizeMask = 0x0F;
Mask for byte size.

nBTFTTypeMask = 0xF0;
Mask for type.

nBTFRducedInt = 0x10;
Int with limited range.

nBTFOunsignedInt = 0x20;
Int with unsigned range.

nBTFSignedInt = 0x40;
Int with signed range.

nBTFFReal = 0x80;
Real.
```

18.10 Masks and values for tstOptions.bFlags

```
nOptTypeMask = $03;
```

In each case 2 bit for the option type.

```
abOTShift: array[0..2] of Byte = (0,2,4);
```

Shift-Faktor for accessing the OptType bits.

```
nOTNone = 0;
```

Option nicht angegeben.

```
nOTInt8 = 1;
```

Option stated as Int8.

```
nOTByte = 2;
```

Option stated as Byte.

```
nOTChar = 3;
```

Option stated as Char.

```
nOFVarParamMask = $40;
```

Var parameter, i.e. address is passed

Attention:

The parameter is not a "real" Var parameter if nOFConstParamMask is set also, i.e. you are not allowed to modify it! Constant Char strings are passed that way for instance.

A dereferenced pointer can point to the const segment without having set the nOFConstParamMask!

```
nOFConstParamMask = $80;
```

Constant.

19 Special data types and structures in pool.pli

Taken from the pool.pli library.

Introduction

This chapter gives a brief description of the special data types of the pool.pli library. For details on how to use these structures refer to the description fo the according functions using the structures.

19.1 File modes

Constants for FMode

The constants specify the file access mode. You can use the modes with binary and text files (see also chapter 6.2).

tenFMode

type

```
tenFMode = (nenFMNone, nenFMRead, nenFMUpdate, nenFMWrite,
            nenFMAppend);
```

Entsprechung in C:

FOpen:	"rb"	"r+b"	"w+b"	"a+b"
TOpen:	"rt"	"r+t"	"wt"	"a+t"

Beschreibung:

nenFMNone Do not open file.

nenFMRead Open file with read access.

nenFMUpdate Open existing file for update. Read and write access with the file pointer pointing to the beginning without previously deleting the file content.

nenFMWrite Create new file or open existing with write access. A possibly existing file get deleted and the data is written at the beginning of the file.

nenFMAppend Create new file or open existing with write access. New data is appended to the existing data.

19.2 Data type QWord

```
QWord
QWord = record      {general unsigned 64 bit data type}
  ?dwLow:   DWord; {remarks: Since the compiler will support this data type
                  in future, the type prefix ("tst") is omitted}
  ?dwHigh:  DWord; {and the elements are not declared public!}
end;
tpQWord = ^QWord;
```

19.3 Types for general library functions

```
tHandle
type
  tHandle   = ^?stHandle;    {(has to be able to handle also a DWord type)}
  tpHandle  = ^tHandle;

tstLibDescr
type
  tstLibDescr = record
    hLibHandle: tHandle;    {library handle}
    pstErrDescr: tpstErrDescr; {information for error handling }
    csLibName:  String;     {library name}
  end;
  tpstLibDescr = ^tstLibDescr;

tstLFDescr
type
  tstLFDescr = record
    pstLibDescr: tpstLibDescr; {library descriptor}
    pFuncAddr:  Pointer;      {the address of the function }
    csFuncName: String;       {name of the function}
  end;
  tpstLFDescr = ^tstLFDescr;
```

19.4 Structure to administrate errors of lib and NoWait functions

```

tstErrDescr
type
  tstErrDescr = record
    pSetErrorFunc: Pointer; {pointer to the function to set error}
    pGetErrorFunc: Pointer; {pointer to the function to retrieve error}
    padwMMO:      tpaDWord; {array with Min/Max/Offset to convert returned
                             error codes to unique values}
  end;
tpstErrDescr = ^tstErrDescr;

```

Beschreibung:

p*ErrorFunc : The functions have to use system thread local error variables!
pSetErrorFunc: Is called, if !=nil, exclusively using the parameter DWord=0 to delete the error code of the "real" function call.
pGetErrorFunc: Is called, if !=nil, after the "real" function call, to calculate the appropriate error code. A DWord is expected as result and OK has to have the value 0!
padwMMO: Array with Min/Max/Offset to convert returned error codes to unique values. This allows a mapping of error sources to error codes. With Min=0, the value is returned without offset.
Min: Minimum error value of the dll.
Max: Maximum error value of the dll.
Offset: Added to the dll value for adjustment purpose.

The returned error codes have to be unique in case that the functions return different kinds of errors (e.g. errno, OSAL and AIDA).

Attention:

The data structure has to fit to the according type definition in the PI!

19.5 Types for system functions

tenNPres: Result for NormPath

type

```
tenNPres = (
  nenNPOk,           {ok, result without "../"}
  nenNPUp,           {result starts with "../" (depends on CWD)}
  nenNPDriveUp,     {result starts with "D:../" (depends on CWD(D))}
  nenNPHostUp,      {result starts with "//Host/Vol/.." (norm. not possible)}
  nenNPRootUp,      {result starts with "../" or "D:../" (not possible if
                    you want the path to be absolute)}
  nenNPOverrun);    {No space available to attach '/' or expand e.g. "/.../"
                    within FILENAME_MAX (i.e. no complete result)}
```

Remarks:

Sorted according to the severity of the error.

tenKinship, nenKS*: Result of GetKinship **{Relationship between records and objects}**

type

```
tenKinship = (
  nenKSNone,         {wrong types (no record or object resp. different type
                    classes)}
  nenKSNoInit,       {at least one type=nil (e.g. object is not initialized)}
  nenKSStrange,      {type1 does not know type2 (i.e. type1 and type2 have no
                    relationship or type1 is an ancestor of type2)}
  nenKSDescendant,  {type1 is descendant of type2}
  nenKSEqual);      {type1 = type2}
```

tenTypeClass

type

```
tenTypeClass = (nenTCNone,          nenTCBoolean,      nenTCChar,
                 nenTCR1,           nenTCR2,           nenTCInteger,
                 nenTCEnum,         nenTCReal,         nenTCR3,
                 nenTCPointer,       nenTCArray,        nenTCCharString,
                 nenTCByteString,    nenTCR4,           nenTCR5,
                 nenTCRecord,        nenTCObject,       nenTCFunction);
```

Remarks:

nenTCFunction:

currently only for debug data and type information from symbol files.

tenBase

type

```
tenBaseType = (nenBT7Bit,          nenBTInt8,         nenBTByte,
                nenBT15Bit,        nenBTInt16,        nenBTWord,
                nenBT31Bit,        nenBTInt32,        nenBTDWord,
                nenBTR1,           nenBTReal32,       nenBTReal64,
                nenBTNOne);
```

Important: Do not use conditions like `X >= nenBTNOne`, since `nenBTNOne` will be set to 0 in a subsequent version!

Commander variables

```

csArg:          String;
Parameterstring from LoadTask or PI command line (-arg, e.g. in Commander:
core.run,0,pi.exe,, "-arg:\-a -b\" test.pi").

boCmdrCommandQuiet: Boolean;
true: Suppress Commander error messages (IDCommand resp. CmdrCommand).

static
  csPromptStr: String = ">";
Prompt for POOL command line.

fiRecordConOut: tFile;
Protocol of all outputs.

```

19.6 Alternative data type denominations

```

String
type
  String = CharString;

```

```

?t15Bit
type
  ?t15Bit

```

Remarks:
 Compiler internal data type, that is used as function result in functions (only in pool.pli), which has to be compatible to DWord and also to Int32. I.e. The result value can be assign to variables of this type. The range is 0..Int32_MAX.

```

tSize
type
  tSize      = Int32;

```

Remarks:
 Int32 is used for sizes and lenght, instead of DWord, since calculations would need type casts otherwise (mixed types would require casts to the next bigger signed type which does not exist). Furthermore some functions return -1 to indicate errors for example.

```

tFOffs
type

```

```

    tFOffs = Int32;
File position for Fseek.

```

tenFOrg

```
type
```

```

    tenFOrg = (nenSeekSet, nenSeekCur, nenSeekEnd);
Origin parameter for Fseek.

```

tenTZ

```
type
```

```

    tenTZ = (nenTZUTC, nenTZLocal);
    nenTZUTC:    UTC (coordinated universal time).
    nenTZLocal:  local time.

```

Short String = null terminated array of s(h)ort Char (like used in C)

```
type
```

```

    tHString    = array[0..] of Char;           {type identifier: hs}
    tpHString   = ^tHString;                   {type identifier: phs}
    tapHString  = array[0..] of tpHString;     {type identifier: ahs}
    tpapHString = ^tapHString;                 {type identifier: pahs}

```

Remarks:

tHString and tpHString are treated special in Write(ln) and Strf, in that they are printed as text instead of Char array.

Attention:

- tpHString is only printed as text in the invariant part, since the required dereferencing would lead to a system crash otherwise!
- C does not differentiate between the array itself and a pointer to the array, whereas POOL does. hsValue from C can be named also phsValue.
- Since POOL does not have any context dependent strings (wide Strings for UNICODE-environments), xs-Variablen from C possibly are named as hs- or phs-variables (e.g. in the AIDA module).

Additional types without further description:

```
type
```

```

    tpBool      = ^Boolean;
    tpChar      = ^Char;
    tpInt8      = ^Int8;
    tpByte      = ^Byte;
    tpInt16     = ^Int16;
    tpWord      = ^Word;
    tpInt32     = ^Int32;
    tpDWord     = ^DWord;
    tpReal32    = ^Real32;
    tpReal64    = ^Real64;
    tpPtr       = ^tpPtr;
    tpCharStr   = ^CharString;
    tpByteStr   = ^ByteString;

    taByte      = array[0..] of Byte;
    tpaByte     = ^taByte;

    taChar      = array[0..] of Char;
    tpaChar     = ^taChar;

```

```
tapaChar = array[0..] of tpaChar;
tpapaChar = ^tapaChar;
tacXlat = array[Char] of Char;
tpacXlat = ^tacXlat;

taPtr = array[0..] of Pointer;
tpaPtr = ^taPtr;

taDWord = array[0..] of DWord;
tpaDWord = ^taDWord;

tabDWord = array [0..SizeOf(DWord)-1] of Byte;
tabWord = array [0..SizeOf(Word)-1] of Byte;
tabInt32 = array [0..SizeOf(Int32)-1] of Byte;
tabInt16 = array [0..SizeOf(Int16)-1] of Byte;
tabReal64 = array [0..SizeOf(Real64)-1] of Byte;
tabReal32 = array [0..SizeOf(Real32)-1] of Byte;
tabPointer= array [0..SizeOf(Pointer)-1] of Byte;

abBTFlags: array[tenBaseType] of Byte = (
  nBTFRducedInt | SizeOf(?t7Bit),
  nBTFSignedInt | SizeOf(Int8),
  nBTFUNsignedInt | SizeOf(Byte),
  nBTFRducedInt | SizeOf(?t15Bit),
  nBTFSignedInt | SizeOf(Int16),
  nBTFUNsignedInt | SizeOf(Word),
  nBTFRducedInt | SizeOf(?t31Bit),
  nBTFSignedInt | SizeOf(Int32),
  nBTFUNsignedInt | SizeOf(DWord),
  0,
  nBTFRReal | SizeOf(Real32),
  nBTFRReal | SizeOf(Real64),
  0);
```

20 Record types

Taken from the pool.pli library.

20.1 Record for keyboard input

```

tunKeyVal
type
  tunKeyVal = record
    variant
      (w: Word);
      {$ifdef BigEndian}
        (enScanCode: tenScanCode; {key code}
         cASCII:      Char);      {ASCII value or '\0' for special keys}
      {$else}
        (cASCII:      Char;
         enScanCode: tenScanCode);
      {$endif}
    end;

```

20.2 Record for system properties

```

tstPIInfo
type
  tstPIInfo = record
    csName:      String;  {"POOL Runtime System"}
    bGeneration: Byte;    {PI's generations number}
    bMajorRel:   Byte;    {major version}
    bMinorRel:   Byte;    {minor version}
    bServiceRel: Byte;    {service release number}
    wYear:       Word;    {PI build date: year}
    bMonth:      Byte;    {PI build date: month}
    bDay:        Byte;    {PI build date: day}
    csCopyright: String;  {PI copyright}
    csOS:        String;  {name of the operating system}
    csOSVersion: String;  {version of the operating system}
    csCmdrHostName: String; {host name of the Commander or '' (no DNS
                             answer), e.g. "bsk-wst-025" }
    csCmdrHostAddr: String; {host address of the Commander as string,
                             e.g. "127.0.0.1"}
  end;

```

20.3 Records for untyped parameters and variable parameter lists

tunEventVal: Optional value of POOL events

```

type
  tunEventVal = record
    variant
      (dw:   DWord);           {unsigned value}
      (i32:  Int32);          {signed value}
      (pv:   Pointer);        {pointer to data}
    end; {tunEventVal}
  tpunEventVal = ^tunEventVal;

```

tstVAInfo

```

type
  tstVAInfo = record
    stVA:      tstUArg;        {type, Opt und pointer to the value (also
                               with VAL-Parameter).}
    ?bVANL:    Byte;          {current nesting level}
    ?b1,?b2,?b3: Byte;        {reserve and padding bytes}
    ?apstVASTack: array[0..?nVAMaxNesting-1] of tpstVArg;
  end;
  tpstVAInfo = ^tstVAInfo;

```

Description:

The structure `tstVAInfo` contains information to the corresponding parameter of a variable parameter list. The contained structure `stVA` of type `tstUArg` contains the details.

tstUArg

```

type
  tstUArg = record
    pstType: tpstType;        {untyped VAR parameter}
    stOpt:   tstOptions;      {pointer to type description}
    pvData:  Pointer;         {options tu current parameter}
  end;                          {pointer to the value (always VAR!)}
  tpstUArg = ^tstUArg;

```

Description:

The structure `tstUArg` contains information to the data type (see also `pstType`), the options (see also `stOpt`) and the data of the untyped parameter.}

tstVArg

```

type
  tstVArg = record
    pstType: tpstType;        {VarArg parameter}
    stOpt:   tstOptions;      {pointer to the type description}
    unData:  tunSTData;       {optionen to the actual parameter}
  end;                          {value or pointer to the variable }

```

```
tDummyArg = record end;    {dummy record}
tpDummyArg = ^tDummyArg;  {type for optional parameter}
```

Description:

The structure `tstVArg` contains information to the data type (see also `pstType`), the options (see also `stOpt`) the data of a variable parameter list.

tstType

```
tstType = record(?tTRec0)    {general type description}
  w0:      Word;             {reserve (e.g. for alignment, shift for
                             bit variables etc.) and alignment}
  csName: String;           {name of the data type}
  xoSize: tSize;           {size in bytes}

  variant
    (stB: tstBase);        {base data type: scalar and subrange types}
    {---}                  {Real is defined by BTyp uniquely}
    (stP: tstPointer);     {pointer type: the type the pointer points
                             to}
    (stA: tstArray);       {array type of selector and component
                             types}
    {---}                  {type string has no obvious structure!}
    (stR: tstRecord);      {record and object type: list of
                             components}

  end;
```

Description:

The structure contains a general description of the data type. The structure is used in conjunction with the `TypeOf` function and in structures to work with variable parameter lists and untyped parameters.

tstOptions

```
type
  tstOptions = record
    bFlags: Byte;           {3*2 bits OptType (nOT*) and
                             nOF*ParamMask}
    variant                 {options as Int8, Byte or Char}
      (ai8Val: array[0..2] of Int8);
      (abVal: array[0..2] of Byte);
      (acVal: array[0..2] of Char);
    end;
```

Description:

The structure contains options respective the parameter in a variable parameter list. A description of `bFlags` is provided in the constants section.

tstVal

```
type
  tstVal = record
    enBType: tenBaseType;   {base type or nenBTNNone in case of an error}
    bRadix: Byte;          {radix to use or 0 for Real}
    variant
      (i32: Int32);        {for all signed and unsigned integert except
                             nenBTWord}
```

```

(dw: DWord);           {for all unsigned integers}
(r64: Real64);        {for real numbers (nenBTReal64)}
(unOrdinal: tunOrdinal); {i32 and dw as tunOrdinal again}
end;

```

Description:

The structure contains the result of the GetVal function (see description of the function). bRadix is used to specify the base to use (0 (for Real), 2, 10 or 16).

tpstVArg.

Record that can contain the data of all simple types.

```

tpstVArg = ^tstVArg;
  tpunSTData = ^tunSTData;
  tunSTData = record
    variant
      (c:      Char);           {Char (Chr(0)..Chr(255)}
      (bo:     Boolean);        {Boolean (false..true)}
      (i8:     Int8);           {Int8 (8 bit signed)}
      (b:      Byte);           {Byte (8 bit unsigned)}
      (i16:    Int16);          {Int16 (16 bit signed)}
      (w:      Word);           {Word (16 bit unsigned )}
      (i32:    Int32);          {Int32 (32 bit signed)}
      (dw:     DWord);          {DWord (32 bit unsigned)}
      (pv:     Pointer);        {untyped pointer}
      (r32:    Real32);          {Real32 (4 byte Real)}
      (r64:    Real64);          {Real64 (8 byte Real)}
      (stCStrHdr: _tstCharString); {CharString header}
      (stBStrHdr: _tstByteString); {ByteString header}
      (punSTD:  tpunSTData);     {pointer to this structure}
      (pstVOT:  tpstType);       {Aktueller Typ von Object-Variablen mit
                                   VMT (Virtual Method Table)}
      (pstVA:   tpstVArg);       {pointer to list for nested lists}
      (pPtr:    tpPtr);          {pointer to itself}
    end;

```

20.4 Record for files (Filehandle)

```

tFile
type
  tFHandle = Word;           {general file handle}
  tFile = record            {file}
    xFHandle: tFHandle;
    bFFlags: Byte;
    enFMode: tenFMode;
    csFName: String;
  end;
  tpFile = ^tFile;

```

20.5 Records for time and date functions

```

tstTZInfo
type
  tstTZInfo = record       {Information to the current time zone}

    acsNames: array[0..1] of String; {name of the current time zone [0]
                                      (never empty) and name of the current
                                      daylight saving [1] (if i8IsDst<=0
                                      possibly empty).}

    i16MinutesWest:      Int16;      {Offset of the local time to UTC in
                                      minutes: local time + i16MinutesWest
                                      = UTC (e.g. 0:00 UTC = 1:00 MET
                                      -> i16MinutesWest = -60)}

    i8IsDst:             Int8;       {DST (daylight saving time) Status:
                                      -1=unknown, 0=off, +1=aktiv.
                                      DST = local time + 60Min (e.g.
                                      0:00 MET = 1:00 MES (DST) resp. DST +
                                      (i16MinutesWest-60) = UTC)}

    bDummy:             Byte;       {Reserve}
    dwTime:             DWord;      {reference time to TZInfo
                                      (UTC like with GetTime_s)}

  end;

```

```

tstDateTime
type
  tstDateTime = record
    wYear:      Word;
    bMonth:     Byte;
    bDay:       Byte;
    bHour:      Byte;
    bMin:       Byte;
    bSec:       Byte;
    bSec100:    Byte;
    wYear:      Word;
    bWeek:      Byte;
    bWDay:      Byte;
    wYDay:      Word;
    i8IsDst:    Int8;
    enTZ:       tentZ;
  end;
  tpstDateTime = ^tstDateTime;
    {date and time}
    {year (including century). Für normale
    Time-Functions currently 1970
    (second 0) to 2038
    (second INT32_MAX)}
    {month (1..12)}
    {day (1..31)}
    {hour (0..23)}
    {minute (0..59)}
    {second (0..60, possibly switch second)}
    {1/100 second (0..99), always 0 in case
    of normal time functions}
    {Year to the week}
    {week according to ISO 8601 (1..53,
    some year only have 52 weeks. Week 1 is
    the first week having 4 days)}
    {weekday according to ISO 8601
    (1..7, 1=Mo)}
    {day of the yahr (0..365, 0=1.Jan)}
    {DST status (see stTZInfo.i8IsDst)}
    {time zone}

```

20.6 Miscellaneous records

```

tunOrdinal
tunOrdinal = record           {ordinal number (with or without sign)}
  variant
    (i32: Int32);             {value of all signed types}
    (dw: DWord);             {value of all unsigned types}
end;
tpunOrdinal = ^tunOrdinal;

tstField
tstField = record             {record and object type: list of components}
  csName:      String;        {name of the component}
  pstType:     tpstType;      {type of the component}
  xoOffs:      tSize;         {offset of the component in the structure}
  wFlags:      Word;          {bFlags, see nFFVariantMask, ..}
end;
tpstField = ^tstField;
tastFields = array[0..] of tstField;
tpastFields = ^tastFields;

tstEnum
tstEnum = record              {enumeration types: list of name and value}
  csName:      String;        {name of the enum constants}
  unValue:     tunOrdinal;     {value of the enum constants}
end;
tpstEnum = ^tstEnum;
tastEnums = array[0..] of tstEnum;
tpastEnums = ^tastEnums;

tstBase
tstBase = record              {base data type: scalar and subrange type}
  unLow:       tunOrdinal;     {lower limit of the allowed range}
  unHigh:      tunOrdinal;     {upper limit of the allowed range}
  pastEnums:  tpastEnums;     {list of the enum constants or nil}
end;
tpstBase = ^tstBase;

tstPointer
tstPointer = record           {pointer type: type the pointer points to}
  pstTarget:   tpstType;       {the type the pointer points to}
end;
tpstPointer = ^tstPointer;

```

```

tstArray
tstArray = record                                {array type of selektor and component
    pstSelectorType:tpstType;                    {selektor type (is always a base type)}
    pstFieldType:  tpstType;                    {component type (can also be an array)}
end;
tpstArray = ^tstArray;

tstRecord
tstRecord = record                              {record and object type: list of components}
    pstParentType:tpstType;                    {the type, the current type inherits from}
    pastFields:  tpastFields;                  {list of the components}
end;
tpstRecord = ^tstRecord;

?tTRec
?tTRec0 = record                               {header for type description}
    enTypeClass:tenTypeClass;                {kind of the data type
                                           {nenTCBoolean,nenTCChar,..}}
    variant
        (enBType:tenBaseType);              {base type of the current type}
        (bFlags: Byte);                     {flags, see TFStrsContMask,..}
end;

_tstCharString
_tstCharString = record                       {Anmerkung: fields in _tstCharString are
    pacCont:  tpaChar;                       {pointer to content (always containing an
    dwFlgLen:  DWord;                         {length and flag}
end;

_tstByteString
_tstByteString = record                      {remark: fields in _tstByteString are only
    pabCont:  tpaByte;                       {pointer to content (always containing an
    dwFlgLen:  DWord;                         {length and flag}
end;

tstTimeVal = record
tstTimeVal = record                          {64 bit time values}
    dwNSec:  DWord;                           {ns (possibly. + flags, see AIDA-Lib)}
    dwMSec:  DWord;                           {ms}
end;
tpstTimeVal = ^tstTimeVal;
tstBool16 = record
tstBool16 = record                           {help construction for a Boolean in 16 bit}
    {$ifdef BigEndian}
    bo1,bo: Boolean;
    {$else}
    bo,bo1: Boolean;

```

```

    {$endif}
end;                                {tstBool16}

tstBool32
tstBool32 = record                  {help construction for a Boolean in 32 bit}
  {$ifdef BigEndian}
    bo3,bo2,bo1,bo: Boolean;
  {$else} {$ifdef LittleEndian}
    bo,bo1,bo2,bo3: Boolean;
  {$else} {$ifdef MiddleEndian}
    bo2,bo3,bo,bo1: Boolean;
  {$endif} {$endif} {$endif}
end; {tstBool32}

tunWord
tunWord = record
  variant
    (i16I: Int16);
    (wW: Word);
    {$ifdef BigEndian}
      (bB1,bB0: Byte);
    {$else}
      (bB0,bB1: Byte);
    {$endif}
    (abA: array[0..1] of Byte);
end; {tunWord}

tunDWord
tunDWord = record
  variant
    (dwD: DWord);
    (i32I: Int32);
    {$ifdef BigEndian}
      (wW1,wW0: Word);
      (i16I1,i16I0: Int16);
      (bB3,bB2,bB1,bB0: Byte);
    {$else} {$ifdef LittleEndian}
      (wW0,wW1: Word);
      (i16I0,i16I1: Int16);
      (bB0,bB1,bB2,bB3: Byte);
    {$else} {$ifdef MiddleEndian}
      (wW1,wW0: Word);
      (i16I1,i16I0: Int16);
      (bB2,bB3,bB0,bB1: Byte);
    {$endif} {$endif} {$endif}
    (abA: array[0..3] of Byte);
end; {tunDWord}

```

21 Shared memory functions (shm.pli)

Taken from the shm.pli library

Introduction

Since the operating system runs each process in its own virtual memory area, to which no other process has access, a common access to variables and thus a direct transmission of data between different processes or tasks is not possible as a matter of principle (see also the description of tasks section). Therefore, special functions are needed for communication between processes. They provide a memory area for the access from various applications (shared memory).

The shm.pli POOL library provides general shared memory functions using the native operating system functions. These functions can be used for both, communication between individual POOL tasks and between POOL tasks and any other (OSAL) application.

To reserve this type of memory area, it first has to be created using the SHM_pstCreate function. It is assigned the desired ID, through which it can be accessed by all tasks or processes.

The memory area can be opened for reading and writing using the ID within the task. To synchronize writing and reading of data in the SHM block (shared memory block) it has to be locked (SHM_boLock, SHM_boTryLock or SHM_boLock_NW). Next, the memory area can be accessed using the MemMove function. After it was accessed it has to be unlocked again in order to allow other processes to use the memory area. After new data has been written, an event can be triggered through the SHM_boSetEvent function, which is used to signal that there is new data. The occurrence of events is detected using the SHM_boWaitEvent or SHM_boWaitEvent_NW functions within the task, and then the read access is started. To do this, the memory area is locked again, the data is read, and the memory is unlocked again.

The ID can be transmitted to the task in two different ways. One way is to transmit the ID using command line parameters (in POOL via csArg). The other way is to use the actual task ID (the task handle) as a SHM ID.

To do this, the task has to be started from another process, which then uses the task ID it receives during the LoadTask call as the ID for the new SHM area. The actual task uses its own task ID to access the SHM.

If the SHM block is no longer needed in an application, it is freed again using the SHM_boClose function. While doing this it has to be observed that the memory is not actually freed until all applications that are using the memory have successfully called the function.

21.1 Types and structures in shm.pli

SHM_tenMode

type

```
SHM_tenMode = (  
  SHM_nenCreate,          {create new shared memory block}  
  SHM_nenOpen,           {open existing shared memory block}  
  SHM_nenOpenExclusive); {open existing shared memory block exclusive}  
                          {(i.e. the created shared memory block can be}  
                          {opened by exactly one other process}
```

Description:

Modes for the access of the shared memory functions.

SHM_tstHandle

type

```
SHM_tstHandle = record  
  pabData: tpaByte;      {pointer to an open array of the data type byte  
                          (tpaByte defined in pool.pli)}  
                          {(additional fields are for internal use only)}  
  
end;  
SHM_tpstHandle = ^SHM_tstHandle;
```

Description:

Handle to the shared memory block.

SHM_pstCreate

21.2 Shared memory functions

Declaration

```
function SHM_pstCreate(xoSize: tSize; dwID: DWord;  
                      enMode: SHM_tenMode): SHM_tpstHandle;
```

Parameter

xoSize.

tSize equals Int32. xoSize defines the size of a newly created shared memory block. To open an existing shared memory block you have to specify the size. The passed size has to match the actual size to avoid undefined behaviour.

dwID.

DWord. Unique ID (1..DWORD_MAX-1), to identify and access the shared memory block from different processes.

enMode.

SHM_tenMode.

Modes:

SHM_nenCreate: Create new shared memory block.

SHM_nenOpen: Open existing shared memory block.

SHM_nenOpenExclusive: Open existing shared memory block exclusively
(i.e. the shared memory block can be opened by exactly one other process.)

Return value

SHM_tpstHandle.

Handle of the created or opened shared memory block (see also 21.1). The return value is nInVHandle case of an error.

Description

The function creates or opens a shared memory block of the size xoSize and returns a handle to it. The return value is nInVHandle case of an error. Error can occur if invalid parameters are passed or if the block already exists and SHM_tenMode = SHM_nenCreate.

SHM_pstCreate**Remarks**

You can set `xoSize` to 0 (e.g. if you only want to use events). Anyway the system allocates memory for internal administration. Since the operating system can allocate memory only in big chunks and in a limited number, it is not useful to use many little memory blocks.

You can open a shared memory block multiple times from within a POOL task or an application, but you can not use the same handle within multiple threads concurrently.

It is useful to use the task handle as shared memory handle if a task is started from a program. `hOwnTHandle` is then used to open the shared memory (since the task handle is the shared memory ID) within the task itself.

Alternatively you can try different IDs (e.g. using `Random`) and pass the used ID via the command line to other processes (see `csArg` for POOL tasks). Conflicts with other applications (non OSAL or POOL) are avoided as possible by additional internal processing, so that you can set fix IDs (e.g. using the command line or a configuration file) in Client/Server applications for instance.

Example

Remarks: The following example shows the communication between a main program and a task that is started by the main program. The main program communicates data via shared memory to the task who puts the data to the standard output. See chapter 11.2 for a description of the task function.

The program was kept simple intentionally to show the basic principle of the SHM functions. Your own applications need appropriate error handling and synchronization of the data transfer.

Main program:

```

module Test;

import SHM;                                {import the SHM library}

function vGetTask: Boolean;
procedure vSendData(bDataPar: Byte;boExitPar: Boolean);

private
  type
    pstShmData = ^tstShmData;              {pointer to the structure}
    tstShmData = record                    {structure to be passed}
      bData: Byte;                          {data to send}
      boExit: Boolean;                       {flag to terminate task}
    end;

var
  hSWHandle: tTHandle;                      {task handle}
  i32Test: Int32;                            {for testing the address range }
  pstShm: SHM_tpstHandle;                    {shared memory handle}
  stShmData: tstShmData;                     {for data to be exchanged}

```

SHM_pstCreate

```

{task loading procedure}
function vGetTask: Boolean;
begin
  vGetTask := false;

  {load task}
  hSWHandle := LoadTask("TaskTest","",0,0);

  if hSWHandle = nil then          {loading failed?}
    Writeln("Could not load task - ",GetErrorMsg(GetError));
    return;                       {return false}
  endif;

  {create a shared memory block, passing task handle as Id}
  pstShm := SHM_pstCreate (SizeOf(tstShmData), DWord(hSWHandle),
                          SHM_nenCreate);

  if pstShm = nil then            {in case creating SHM fails }
    Writeln("Could not open shared memory - ",GetErrorMsg(GetError));
    return;                       {return false}
  endif;

  if not StartTask(hSWHandle) then {start tast }
    WritelnStr("Could not start the task - ",GetErrorMsg(GetError));
    return;                       {return false}
  endif;

  vGetTask := true;              {return true}
end;

procedure vSendData(bDataPar: Byte; boExitPar: Boolean);
begin
  {lock shared memory block}
  if !SHM_boLock(pstShm) then
    Writeln("Could not lock shared memory block - ",GetErrorMsg(GetError));
    return;
  endif;

  stShmData.bData := bDataPar;    {write data to shared memory}
  stShmData.boExit := boExitPar;  {set exit flas}

  {copy object data to shared memory block}
  MemMove(stShmData,pstShm^.pabData^,SizeOf(tstShmData));

  {set data available event}

  if !SHM_boSetEvent(pstShm) then
    Writeln("Could not send event - ",GetErrorMsg(GetError));
    return;
  endif;

  {unlock shared memory block}
  if !SHM_boUnlock(pstShm) then
    Writeln("Could not unlock shared memory block - ",GetErrorMsg(GetError));

```

SHM_pstCreate

```

    return;
endif;

end;

{main program}
procedure vMain;
var
    bData: Byte;

begin
    if vGetTask then                {start task and create SHM}
        repeat
            bData := bData + 1;    {datan to pass}
            vSendData(bData,false); {communicate data via SHM
                                     (every second (1000ms))}

            Writeln("Main program\n");
            Wait(1000);

            until (KeyPressed <> 0) {terminate program when key is hitted}
            vSendData(0,true);      {send command to terminate task}
        endif;
    end;

{vDeinit: deinitialize module, task, and SHM}
procedure vDeinit;
var
    i32SWExitCode: Int32;
begin
    {wait until task has been terminated }
    while GetThreadState(hSWHandle) > nenTSTerm do
        Wait(25);
    endwhile;

    i32SWExitCode := GetThreadExit-Code(hSWHandle);
    if i32SWExitCode <> 0 then
        Writeln("Exit-Code",i32SWExitCode);
    endif;

    {unload the task}
    if (UnloadTask(hSWHandle)) then
        Writeln("Unloading task failed - ",GetErrorMsg(GetError));
    endif;
    if !SHM_boClose(pstShm) then      {close SHM}
        Writeln("Error while closing SHM - ",GetErrorMsg(GetError));
    endif;
end;

{module initialisation}
begin
end.

```

SHM_pstCreate**Task:**

```

module TaskTest;
import SHM;

private
  type
    pstShmData = ^tstShmData;
    tstShmData = record
      bData: Byte;
      boExit: Boolean;
    end;
    {define structure}
    {data to send}
    {flag to terminate task}

var
  pstShm: SHM_tpstHandle;
  stShmData: tstShmData;
  {handle to shared memory block}
  {structure of the data to receive}

procedure vMain;
begin
  {open the existing shared memory block}
  pstShm := SHM_pstCreate(
    SizeOf(tstShmData), DWord(hOwnTHandle),
    SHM_nenOpenExclusive);

  if pstShm = nil then
    {in case opening is not possible}
    Writeln("Task-Error: Could not open SHM - ", GetErrorMsg(GetError));
    return;
  endif;

  repeat
    if !(SHM_boWaitEvent(pstShm)) then {new data available?}
      Writeln("Wait event failed - ", GetErrorMsg(GetError));
      return;
    endif;
    Writeln("read data");
    if !(SHM_boLock(pstShm)) then {lock SHM}
      Writeln("Task-Error: lock failed - ", GetErrorMsg(GetError));
      return;
    endif;
    {read objectdata from shared memory block}
    MemMove(pstShm^.pabData^, stShmData, SizeOf(tstShmData));

    if !(SHM_boUnlock(pstShm)) then {unlock SHM}
      Writeln("Task-Error: unlock failed");
      return;
    endif;
    Writeln(stShmData.bData);
  until (stShmData.boExit);
  {print data}
  {exit if exit flas was set}
end;

begin
end.

```

SHM_boClose**Declaration**

```
function SHM_boClose(var pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

pstHandle.

SHM_tpstHandle as reference. Handle to the shared memory to close.

Return value

Boolean.

true if successful, otherwise false.

Description

The function closes a shared memory block that is identified by the passed handle. The function returns true if successful and false in case of an error (e.g. an invalid handle).

Remarks

The handle is always deleted when calling the function SHM_boClose. The shared memory block itself is deleted after the last instance pointing to it is deleted.

Example

(see also SHM_pstCreat).

```
if !SHM_boClose(pstShm) then           {close SHM}
  Writeln("Error during closing the SHM-",GetErrorMsg(GetError));
endif;
```

SHM_boLock**Declaration**

```
function SHM_boLock(pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

pstHandle.

SHM_tpstHandle. Handle to the shared memory block to lock.

Return value

Boolean.

true if successful, otherwise false.

Description

The function locks the shared memory block identified by the passed handle. The function waits if the shared memory is already locked. The function returns true if successful and false in case of an error (e.g. an invalid handle).

Remarks

You have to acquire a lock on the shared memory block before you can access it!

You have to call Unlock exactly once after a successful lock operation, before the next lock operation is allowed.

The function waits for an Unlock if another task has already locked the shared memory block.

Example

```
{lock shared memory }
if !SHM_boLock(pstShm) then
  Writeln("Unable to lock - ",GetErrorMsg(GetError));
  return;
endif;
```

SHM_vLock_NW

Declaration

```
procedure SHM_vLock_NW(var oNWEvent: toNWEvent;  
                       pstHandle: SHM_tpstHandle);
```

Parameter

oNWEvent.

toNWEvent as reference. Event class of the NoWait objects. The object has to be initialized using the constructor polnit before it is used and it has to be deleted using vDone before the program terminates.

The object contains the error code (oNWEvent.dwErrNo) after the event is signaled the result of the function call (oNWEvent.unVal.dw, oNWEvent.unVal.i32). For details see the description of the toEvent object.

pstHandle.

SHM_tpstHandle. Handle to the shared memory block to lock.

Return value

None.

Description

The procedure locks the shared memory block identified by the passed handle. The function waits if the shared memory is already locked. Locking the shared memory after another process has unlocked the shared memory triggers an NoWait event (see the description of the functions WaitEvent/WaitEvents) you can react on. Errors can occur e.g. when passing invalid parameters.

Remarks

You have to acquire a lock on the shared memory block before you can access it!

You have to call Unlock exactly once after a successful lock operation, before the next lock operation is allowed.

The procedure waits for an Unlock if another task has already locked the shared memory block.

Example

See also the example to SHM_pstCreat.

SHM_boTryLock

Declaration

```
function SHM_boTryLock(pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

pstHandle.

SHM_tpstHandle. Handle to the shared memory block to lock.

Return value

Boolean.

`true` if successful . `false` in case of an error like invalid handle or shared memory is already locked by another task.

Description

The function tries to lock a shared memory. `false` is returned if the function fails (the shared memory is already locked or an invalid handle was provided) and `true` if the function succeeded in locking the shared memory.

Remarks

The function returns immediatly without waiting for the event like `vLock`.

You have to acquired a lock on the shared memory block before you can access it!

You have to call `Unlock` exactly once after a successful lock operation, before the next lock operation is allowed.

The function waits for an `Unlock` if another task has already locked the shared memory block.

Example

```
{lock shared memory block }
if !SHM_boTryLock(pstShm) then
  Writeln("Could not lock shared memory - ",GetErrorMsg(GetError));
  return;          {return to the calling program}
endif;
```

SHM_boUnlock

Declaration

```
function SHM_boUnlock(pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

pstHandle.

SHM_tpstHandle. Handle to the shared memory block to unlock.

Return value

Boolean.

true if successful, otherwise false.

Description

The function unlocks a shared memory block that was locked using vLock resp. vTryLock. The function returns true if successful and false otherwise.

Remarks

You have to call Unlock exactly once after a successful lock operation, before the next lock operation is allowed.

The function waits for an Unlock if another task has already locked the shared memory block.

Example

```
{unlock shared memory block }  
if !SHM_boUnlock(pstShm) then  
  WriteLn("Could not unlock shared memory block - ",GetErrorMsg(GetError));  
  return;  
endif;
```

SHM_boSetEvent**Declaration**

```
function SHM_boSetEvent(pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

`pstHandle`.

`SHM_tpstHandle`. Handle to the shared memory block whose event is to be signaled.

Return value

Boolean.

`true` if successful, otherwise `false`.

Description

The function set a shared memory event. The function returns `true` if successful and `false` otherwise.

Remarks

Each block has a (system) event, that can be signaled only once (i.e. it's useless to signal the event multiple times as long as the event did not wake up a process).

Example

```
{set event}
if !SHM_boSetEvent(pstShm) then
  Writeln("Could not set event - ",GetErrorMsg(GetError));
  return;
endif;
```

SHM_boWaitEvent

Declaration

```
function SHM_boWaitEvent(pstHandle: SHM_tpstHandle): Boolean;
```

Parameter

pstHandle.

SHM_tpstHandle. Handle to the shared memory block whose event is to be signaled.

Return value

Boolean.

true if successful, otherwise false.

Description

The function waits for an event, that was signaled by another process. The event can be used synchronise processes communicating via shared memory.

The function returns true if successful and false otherwise.

Remarks

Each block has a (system) event, that can be signaled only once (i.e. it's useless to signal the event multiple times as long as the event did not wake up a process).

Example

```
if !(SHM_boWaitEvent(pstShm)) then {new data available}
  Writeln("Wait-Event failed - ",GetErrorMsg(GetError));
  return;
endif;

Writeln("read data");

if !(SHM_boLock(pstShm)) then      {lock SHM}
  Writeln("Task-Error: lock operation failed - ",GetErrorMsg(GetError));
  return;
endif;

{read object data from shared memory}
MemMove(pstShm^.pabData^,stShmData,SizeOf(tstShmData));

if !(SHM_boUnlock(pstShm)) then  {unlock SHM}
  Writeln("Task-Error: unlock operation failed", GetErrorMsg(GetError));
  return;
endif;

Writeln(stShmData.bData);        {print the data}
```

SHM_vWaitEvent_NW

Declaration

```
procedure SHM_vWaitEvent_NW(var oNWEvent: toNWEvent; pstHandle:
SHM_tpstHandle);
```

Parameter

oNWEvent.

toNWEvent as reference. Event class of the NoWait objects. The object has to be initialized using the constructor polnit before it is used and it has to be deleted using vDone before the program terminates.

The object contains the function result (oNWEvent.unVal.dw, oNWEvent.unVal.i32, or oNWEvent.unVal.pv). For details see the description of the toEvent object.

pstHandle.

SHM_tpstHandle. Handle to the shared memory block whose event is to be signaled.

Return value

None.

Description

NoWait counterpart of the function SHM_vWaitEvent.

A successful call of vWaitEvent returns true, otherwise false is returned.

Remarks

Each block has a (system) event, that can be signaled only once (i.e. it's useless to signal the event multiple times as long as the event did not wake up a process).

Example

(see also the example to SHM_pstCreate)

```
procedure vSendData(bDataPar: Byte; boExitPar: Boolean);
begin
    {lock shared memory block}
    SHM_vLock_NW(oNWEvent, pstShm);

    if WaitEvents(oNWEvent, oTEvent) = tpoEvent(@oTEvent) then
        Writeln("Timeout-vSendData");
        return;
    endif;
    stShmData.bData := bDataPar;           // set data
```

SHM_vWaitEvent_NW

```
stShmData.boExit := boExitPar;          // set exit flag
{copy object data to shared memory }
MemMove(stShmData,pstShm^.pabData^,SizeOf(tstShmData));

{send data available event to task}
if !SHM_boSetEvent(pstShm) then
  Writeln("Could not send event - ",GetErrorMsg(GetError));
  return;
endif;

{unlock shared memory block }
if !SHM_boUnlock(pstShm) then
  Writeln("Could not unlock shared memory - ",GetErrorMsg(GetError));
  return;
endif;
end;
```

Attachment

A1 Bibliography

Books:

[1]

PC-Intern 5 - Systemprogrammierung, M. Tischer, B. Jennrich, DataBecker.

Describes Windows 95 system programming. Everybody who intends to use multithreading and memory management and has little or no experience in that area should read the detailed description of multithreading and virtual memory management thoroughly.

[2]

GoTo - C++-Programmierung, A. Willms, ADDISON-WESLEY.

Introduction into OOP programming. Lots of examples are provided especially working with files.

Tutorials:

POOL-Tutorial part 1 – Basics of the POOL programming language, by T.Locker

Revised by U. Kühn.

Simple introduction to the basics of the POOL programming language, inclusive lots of examples and exercises.

POOL-Tutorial part 2 – OOP and POOL, by T.Locker

Revised by U. Kühn.

Comprehensible introduction to object oriented programming using POOL, inclusive lots of examples and exercises.

A2 Index

- ?nVAMaxNesting 354
- ?t15Bit 366
- AB2BStr 86
- abBTFlags 368
- abOTShift 353, 360
- Abs 297
- acEmptyHStr 352
- AdaptVal 330
- Addr 307
- ArcCos 9
- ArcSin 10
- ArcTan 11
- ArcTan2 12
- Arithmetical functions 8
- Attachment 396
- base class toRoot 136
- bFlags 358, 360
- Bibliography 396
- Bin2Hex 32
- Bin2HexD 33
- Bitmasks for tstField.wFlags 358
- Bitmasks for tstType.bFlags 358
- BlockRead 101
- BlockWrite 103
- Bottom 34
- BoVal 64
- bsEmptyBStr 352
- BStrOf 85
- BVal 60
- BValh 62
- ByteString conversion to strings 32
- CallFunc 287
- CallFunc_NW 290
- Character functions 29
- Character to string conversions 65
- CharString 366
- ChLower 29
- Chr 298
- ChUpper 30
- ClrEol 174
- ClrScr 175
- Commander variables 366
- Compiler internal standard functions 296
- Constant *_MAX 351
- Constants (general) 351
- Constants for control characters 356
- Constants for files n* ** 355
- Constants for FMode 361
- Constants for library functions 358
- Constants for variable parameter lists and untyped parameter 353
- Constants from pool.pli 350
- Constants n* 353
- Constants st* 353
- Conversion using conversation vectors 348
- Copy 36
- Cos 13
- CreateThread 217
- csArg 366
- Data type QWord 362
- data types and structures in pool.pli 361
- DataSize 332
- Date function records 373
- Date functions 269
- Dec 300
- Delete 37
- DelSpace 39
- DestroyThread 221
- Dispose 312, 313
- Dummy Optionen 353
- DWORD_MAX 351
- dwRound 21
- dwTrunc 23
- DWVal 60
- DWValh 62
- Dynamic memory management 307
- Error code *_MAX 351
- Error codes 249
- Error E* 251
- Error functions 242
- ERROR_* 250
- Event handling 148
- Event objects 150
- Exp 16
- ExpEnvVars 256
- Exponential and logarithm functions 16
- false 350
- FClose 100
- FEof 105
- FFlush 107
- FGetPos 115
- File handle 373
- File handle records 373
- File I/O functions 95
- File modes 361
- Files (text files) 117
- FMode 361
- FOpen 98
- Frac 26
- FSeek 109
- FSetPos 113
- FSize 111
- GetError 243
- GetErrorMsg 245
- GetEventNo 168
- GetEventSignaled 170
- GetHDWofQW 93
- GetKinship 341

-
- GetLDWofQW 94
 - GetLibFunction 279
 - GetLine 194
 - GetMaxX 176
 - GetMaxY 177
 - GetThreadExit-Code 225
 - GetThreadHandle 222
 - GetThreadState 223, 234
 - GetTime_ms 269
 - GetTime_s 270
 - GetUAVal 328
 - GetVA 320
 - GetVANext 326
 - GetVAVal 321
 - GetX 178
 - GetY 179
 - GotoXY 180
 - Halt 246
 - HeapBlockSize 334
 - Hex2Bin 87
 - Hexadecimal format conversion 343
 - HexDump 343
 - hOwnTHandle 352
 - HStr2Str 66
 - I16Val 60
 - i32Round 21
 - i32Trunc 23
 - I32Val 60
 - I8Val 60
 - IDCommand 182
 - IDCommand_NW 184
 - Inc 302
 - Insert 40
 - Int 24
 - INT16_MIN 351
 - Int32 366
 - INT32_MAX 351
 - INT32_MIN 351
 - INT8_MAX 351
 - INT8_MIN 351
 - Keyboard and monitor functions 173
 - Keyboard codes 198
 - Keyboard input record 369
 - KeyPressed 187
 - Length 41
 - LibCall 281
 - LibCall_NW 284
 - Library functions 275
 - Ln 17
 - LoadLib 276
 - LoadTask 229
 - Log 18
 - MakeDateTime 271
 - MakeTime_s 273
 - Masks (general) 351
 - Masks and values for tstOptions.bFlags 360
 - Masks from pool.pli 350
 - Masks of error codes 249
 - MemCmp 335
 - MemMove 337
 - Memory functions 332
 - MemSet 339
 - Menu functions 202
 - MenuClose 211
 - MenuCloseAll 212
 - MenuCount 213
 - MenuExecute 206
 - MenuExecute_NW 208
 - MenuShow 203
 - Miscellaneous records 375
 - Monitor and keyboard functions 173
 - Move2Str 83
 - Mutex 235
 - Mutex objects 236
 - nADXErrSrcMask 249
 - nAIDAErrSrcMask 249
 - nBTF* 358
 - nc* 356
 - nenSC* 199
 - New 309, 310, 311
 - NewIn 181
 - nFFCRLFMask 355
 - nFFLFMask 355
 - nFFPresNLMask 355
 - nFFPrivateMask 358
 - nFFVariantMask 358
 - nil 350
 - nInvFHandle 355
 - nInvHandle 358
 - nMaxIDLen 352
 - nMaxOASize 351
 - nOFConstParamMask 353, 360
 - nOFVarParamMask 353, 354, 360
 - nOptTypeMask 353, 360
 - NormPath 261
 - nOSALerr* 253
 - nOSALerrSrcMask 249
 - nOTByte 353, 360
 - nOTChar 353, 360
 - nOTInt8 353, 360
 - nOTNone 353, 360
 - nSockErrSrcMask 249
 - nStdErrSrcMask 249
 - nSysErrSrcMask 249
 - nTFStrsContMask 358
 - nVAMaxNesting 354
 - nwTimeoutInfinite 351
 - Object toRoot 136
 - Optional parameter functions 315
 - Optional parameters 314
 - Ord 299
 - Path2Host 263
 - pi 350
 - POOL- base classes 136
-

-
- Pos 42
 - PVANext 318
 - QWord 362
 - QWord functions 89
 - R32Val 60
 - r64Round 27
 - r64Trunc 25
 - R64Val 60
 - Random 345
 - Random number functions 345
 - Randomize 347
 - ReadKey 191
 - ReadKey_NW 192
 - ReadStr 193
 - Record types 369
 - Relationship of objects 341
 - Round 21
 - SetError 244
 - SetExitCode 248
 - SetHDWofQW 91
 - SetLDWofQW 92
 - SetLibErrDescr 294
 - SetQW 89
 - Shared memory functions 380
 - Shared memory functions (shm.pli) 378
 - Shared memory types and structures in shm.pli 379
 - SHM_boClose 386
 - SHM_boLock 387
 - SHM_boSetEvent 391
 - SHM_boTryLock 389
 - SHM_boUnlock 390
 - SHM_boWaitEvent 392
 - SHM_pstCreate 380
 - SHM_tenMode 379
 - SHM_tstHandle 379
 - SHM_vLock_NW 388
 - SHM_vWaitEvent_NW 394
 - Sin 14
 - SIZE_MAX 351
 - SizeOf 304
 - Sqr 19
 - Sqrt 20
 - st*Info 350
 - Standard C error codes 251
 - Standard constants (internal to the compiler) 350
 - Standard-BSK-OSAL-Error codes 253
 - StartTask 231
 - stDummyTCOpt 354
 - stDummyValOpt 353
 - stDummyVarOpt 353
 - Str2HStr 31
 - StrbB 68
 - StrbDW 68
 - StrBo 67
 - StrbW 68
 - StrCh2Int 58
 - StrCompress 44
 - StrExpand 45
 - Strf 81
 - StrfbDW 74
 - Strfbl32 74
 - StrfDW 72
 - StrfhDW 76
 - Strfl32 72
 - StrfPtr 80
 - StrfRe 78
 - StrfStr 51
 - String and character functions 28
 - String conversions to values 55
 - StrLower 48
 - StrOf 65
 - Structure to administrate errors of lib and NoWait functions 363
 - StrUpper 47
 - StrVal 71
 - SwapStrs 49
 - Synchronisation objects (Mutex) 235
 - System 264
 - System error codes 250
 - System functions 255
 - System properties record 369
 - System_NW 266
 - Tan 15
 - tapHString 367
 - Task functions 214
 - Tasks 228
 - TClose 120
 - tenBase 364
 - tenFMode 361
 - tenFOrg 367
 - tenKinship 364
 - tenNPRes 364
 - tenTypeClass 364
 - tenTZ 367
 - TEof 121
 - TerminateThread 227
 - Text files 117
 - tFile 373
 - TFlush 122
 - tFOffs 367
 - tHandle 362
 - Thread functions 214
 - Threads 214
 - tHString 367
 - Time function records 373
 - Time functions 269
 - toEMRoot 144
 - constructor poInit 146
 - destructor vDone 147
 - toEvent 150
 - boQueued 154
 - dwErrNo 152
-

-
- Methods 151
 - pstValType 152
 - public properties 152
 - tpoEvent 153
 - unVal 152
 - vSignal 155
 - toMutex
 - boTryLock 240
 - constructor poInit 238
 - Methods 237
 - vLock 239
 - vUnlock 241
 - toNWEvent 162
 - constructor poInit 163
 - Methoden 162
 - Top 50
 - TOpen 118
 - toRoot 136
 - constructor 139
 - constructor poReinit 141
 - destructor vDone 142
 - vWriteObj 143
 - toTEvent 156
 - constructor poInit 158
 - Methoden 157
 - vSetDT 159
 - vSetRM 160
 - tpapHString 367
 - tpHString 367
 - tpstVArg 372
 - Transfer functions 21
 - TRead 124
 - TReadln 126
 - Trigonometrical functions 9
 - true 350
 - Trunc 23
 - tstArray 376
 - tstBase 375
 - tstBool16 376
 - tstBool32 377
 - tstDateTime 374
 - tstEnum 375
 - tstErrDescr 363
 - tstField 375
 - tstField.wFlags 358
 - tstLFDescr 362
 - tstLibDescr 362
 - tstOptions 354, 371
 - tstOptions.bFlags 353, 360
 - tstPllInfo 369
 - tstRecord 376
 - tstTimeVal 376
 - tstType 371
 - tstType.bFlags 358
 - tstTZInfo 373
 - tstUArg 370
 - tstVAlInfo 370
 - tstVal 372
 - tstVArg 371
 - tunDWord 377
 - tunEventVal 370
 - tunKeyVal 369
 - tunOrdinal 375
 - tunWord 377
 - TWriteBo 128
 - TWriteCh 128
 - TWriteDW 128
 - TWriteI32 128
 - TWriteInBo 130
 - TWriteInCh 130
 - TWriteInDW 130
 - TWriteInI32 130
 - TWriteInPtr 130
 - TWriteInRe 130
 - TWriteInStr 130
 - TWritePtr 128
 - TWriteRe 128
 - TWriteStr 128
 - Typen ohne nähere Beschreibung 367
 - TypeOf 306
 - Types for general library functions 362
 - Types for system functions 364
 - UAFirst 327
 - unLastKeyVal 351
 - UnloadLib 278
 - UnloadTask 233
 - Untyped parameter functions 327
 - Untyped parameters 314
 - VAFirst 315
 - Val 55
 - Val function 330
 - Value to string conversions 67
 - VANext 323
 - Variable parameter list functions 315
 - VecConv 348
 - VecConvI 348
 - vStrXlat 53
 - Wait 164
 - WaitEvent 165
 - WaitEvents 166
 - WaitKey 188
 - WaitKey_NW 189
 - wFlags 358
 - WORD_MAX 351
 - Write 132
 - Writeln 134
 - WVal 60
 - WValh 62