



Standard Library Reference

Boca Raton Documentation Team

Standard Library Reference

Boca Raton Documentation Team

Copyright © 2016 HPCC Systems®. All rights reserved

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com>

Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

LexisNexis and the Knowledge Burst logo are registered trademarks of Reed Elsevier Properties Inc., used under license.

HPCC Systems® is a registered trademark of LexisNexis Risk Data Management Inc.

Other products, logos, and services may be trademarks or registered trademarks of their respective companies.

All names and example data used in this manual are fictitious. Any similarity to actual persons, living or dead, is purely coincidental.

2016 Version 6.2.0-1

<i>Logical Files</i>	7
CompareFiles	8
DeleteLogicalFile	9
LogicalFileList	10
FileExists	11
ForeignLogicalFileName	12
GetFileDescription	13
GetLogicalFileAttribute	14
ProtectLogicalFile	15
RenameLogicalFile	16
SetFileDescription	17
SetReadOnly	18
VerifyFile	19
<i>SuperFiles</i>	20
CreateSuperFile	21
SuperFileExists	22
DeleteSuperFile	23
GetSuperFileSubCount	24
GetSuperFileSubName	25
LogicalFileSuperOwners	26
LogicalFileSuperSubList	27
SuperFileContents	28
FindSuperFileSubName	29
StartSuperFileTransaction	30
AddSuperFile	31
RemoveSuperFile	32
ClearSuperFile	33
RemoveOwnedSubFiles	34
SwapSuperFile	35
ReplaceSuperFile	36
PromoteSuperFileList	37
FinishSuperFileTransaction	38
<i>External Files</i>	39
ExternalLogicalFileName	40
MoveExternalFile	41
DeleteExternalFile	42
CreateExternalDirectory	43
RemoteDirectory	44
<i>File Browsing</i>	45
SetColumnMapping	46
GetColumnMapping	47
AddFileRelationship	48
FileRelationshipList	50
RemoveFileRelationship	51
<i>File Movement</i>	52
DfuPlusExec	53
AbortDfuWorkunit	54
Copy	55
DeSpray	57
RemotePull	58
Replicate	60
SprayFixed	61
SprayVariable	62
SprayXML	64

WaitDfuWorkunit	65
<i>String Handling</i>	66
CleanAccents	67
CleanSpaces	68
CompareAtStrength	69
CompareIgnoreCase	70
Contains	71
CountWords	72
EditDistance	73
EditDistanceWithinRadius	74
EndsWith	75
EqualIgnoreCase	76
ExcludeFirstWord	77
ExcludeLastWord	78
ExcludeNthWord	79
Extract	80
Filter	81
FilterOut	82
Find	83
FindCount	84
FindAtStrength	85
FindAtStrengthReplace	86
FindReplace	87
FromHexPairs	88
GetNthWord	89
RemoveSuffix	90
Repeat	91
Reverse	92
SplitWords	93
SubstituteExcluded	94
SubstituteIncluded	95
StartsWith	96
ToHexPairs	97
ToLowerCase	98
ToTitleCase	99
ToUpperCase	100
Translate	101
WildMatch	102
WordCount	103
<i>Metaphone Support</i>	104
Primary	105
Secondary	106
Double	107
<i>Date and Time Handling</i>	108
Date Data Types	109
Time Data Types	110
Year	111
Month	112
Day	113
Hour	114
Minute	115
Second	116
DateFromParts	117
TimeFromParts	118

IsLeapYear	119
IsDateLeapYear	120
IsValidDate	121
IsValidTime	122
IsValidGregorianDate	123
FromGregorianYMD	124
ToGregorianYMD	125
FromStringToDate	126
Today	127
CurrentDate	128
CurrentTime	129
DayOfYear	130
DaysBetween	131
MonthsBetween	132
AdjustDate	133
AdjustCalendar	134
<i>Cluster Handling</i>	135
Node	136
Nodes	137
LogicalToPhysical	138
DaliServer	139
Group	140
GetExpandLogicalFileName	141
<i>Job Handling</i>	142
WUID	143
Target	144
Name	145
User	146
OS	147
Platform	148
LogString	149
<i>File Monitoring</i>	150
MonitorFile	151
MonitorLogicalFileName	153
<i>Logging</i>	155
dbglog	156
addWorkunitInformation	157
addWorkunitWarning	158
addWorkunitError	159
<i>Auditing</i>	160
Audit	161
<i>Utilities</i>	162
GetHostName	163
ResolveHostName	164
CmdProcess	165
GetUniqueInteger	166
<i>Debugging</i>	167
GetParseTree	168
GetXMLParseTree	169
Sleep	170
msTick	171
<i>Email</i>	172
SendEmail	173
SendEmailAttachData	174

SendEmailAttachText	175
<i>Workunit Services</i>	176
WorkunitExists	177
WorkunitList	178
SetWorkunitAppValue	180
WUIDonDate	181
WUIDdaysAgo	182
WorkunitTimeStamps	183
WorkunitMessages	184
WorkunitFilesRead	185
WorkunitFilesWritten	186
WorkunitTimings	187
<i>BLAS Support</i>	188
Types	189
ICellFunc	190
Apply2Cells	191
dasum	192
daxpy	193
dgemm	194
dgetf2	195
dpotf2	196
dscal	197
dsyrk	198
dtrsm	199
extract_diag	200
extract_tri	201
make_diag	202
make_vector	203
trace	204

Logical Files

CompareFiles

STD.File.CompareFiles(*file1*, *file2* [, *logicalonly*] [, *usecrcs*])

<i>file1</i>	A null-terminated string containing the logical name of the first file.
<i>file2</i>	A null-terminated string containing the logical name of the second file.
<i>logicalonly</i>	Optional. A boolean TRUE/FALSE flag that, when TRUE, does not compare physical information from disk but only the logical information in the system datastore (Dali). If omitted, the default is TRUE.
<i>usecrcs</i>	Optional. A boolean TRUE/FALSE flag indicating that, when TRUE, compares physical CRCs of all the parts on disk. This may be slow on large files. If omitted, the default is FALSE.
Return:	CompareFiles returns returns an INTEGER4 value.

The **CompareFiles** function compares *file1* against *file2* and returns the following values:

0	<i>file1</i> and <i>file2</i> match exactly
1	<i>file1</i> and <i>file2</i> contents match, but <i>file1</i> is newer than <i>file2</i>
-1	<i>file1</i> and <i>file2</i> contents match, but <i>file2</i> is newer than <i>file1</i>
2	<i>file1</i> and <i>file2</i> contents do not match and <i>file1</i> is newer than <i>file2</i>
-2	<i>file1</i> and <i>file2</i> contents do not match and <i>file2</i> is newer than <i>file1</i>

Example:

```
A := STD.File.CompareFiles('Fred1', 'Fred2');
```


DeleteLogicalFile

STD.File.DeleteLogicalFile(*filename* [, *ifexists*])

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>ifexists</i>	Optional. A boolean value indicating whether to post an error if the <i>filename</i> does not exist. If omitted, the default is FALSE.

The **DeleteLogicalFile** function removes the named file from disk.

Example:

```
A := STD.File.DeleteLogicalFile('Fred');
```

LogicalFileList

STD.File.LogicalFileList([*pattern*] [, *includenormal*] [, *includesuper*] [, *unknownszero*] [, *foreigndali*])

<i>pattern</i>	Optional. A null-terminated string containing the mask of the files to list. If omitted, the default is '*' (all files).
<i>includenormal</i>	Optional. A boolean flag indicating whether to include “normal” files. If omitted, the default is TRUE.
<i>includesuper</i>	Optional. A boolean flag indicating whether to include SuperFiles. If omitted, the default is FALSE.
<i>unknownszero</i>	Optional. A boolean flag indicating to set file sizes that are unknown to zero (0) instead of minus-one (-1). If omitted, the default is FALSE.
<i>foreigndali</i>	Optional. The IP address of the foreign dali used to resolve the file. If blank then the file is resolved locally. If omitted, the default is blank.
Return:	LogicalFileList returns returns a dataset in the following format:

```
EXPORT FsLogicalFileNameRecord := RECORD
  STRING name;
END;

EXPORT FsLogicalFileInfoRecord := RECORD(FsLogicalFileNameRecord)
  BOOLEAN superfile;
  UNSIGNED8 size;
  UNSIGNED8 rowcount;
  STRING19 modified;
  STRING owner;
  STRING cluster;
END;
```

The **LogicalFileList** function returns a list of the logical files in the environment files as a dataset in the format listed above.

Example:

```
OUTPUT(STD.File.LogicalFileList());
//returns all normal files

OUTPUT(STD.File.LogicalFileList(, FALSE, TRUE));
//returns all SuperFiles
```

FileExists

STD.File.FileExists(*filename* [, *physicalcheck*])

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>physicalcheck</i>	Optional. A boolean TRUE/FALSE to indicate whether to check for the physical existence the <i>filename</i> on disk. If omitted, the default is FALSE.
Return:	FileExists returns a BOOLEAN value.

The **FileExists** function returns TRUE if the specified *filename* is present in the Distributed File Utility (DFU). If *physicalcheck* is set to TRUE, then the file's physical presence on disk is also checked.

Example:

```
A := STD.File.FileExists('~CLASS::RT::IN::People');
```

See Also: SuperFileExists

ForeignLogicalFileName

STD.File.ForeignLogicalFileName(*filename* [, *foreigndali*] [, *absolute*])

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>foreigndali</i>	A null-terminated string containing the IP address of the foreign Dali. If omitted, the <i>filename</i> is presumed to be a foreign logical file name, which is converted to a local logical file name.
<i>absolute</i>	Optional. A boolean TRUE/FALSE to indicate whether to prepend a tilde (~) to the resulting foreign logical file name. If omitted, the default is FALSE.
Return:	ForeignLogicalFileName returns returns a VARSTRING (null-terminated) value.

The **ForeignLogicalFileName** function returns either a foreign logical file name (if the *foreigndali* parameter is present) or a local logical file name.

Example:

```
sf := '~thor_data400::BASE::Business_Header';
ff := STD.File.ForeignLogicalFileName(sf, '10.150.29.161', true);
//results in: ~foreign::10.150.29.161::thor_data400::base::business_header
lf := STD.File.ForeignLogicalFileName(ff, '', true);
//results in: ~thor_data400::base::business_header
```

GetFileDescription

STD.File.GetFileDescription(*filename*)

<i>filename</i>	A null-terminated string containing the logical name of the file.
Return:	GetFileDescription returns a VARSTRING (null-terminated) value.

The **GetFileDescription** function returns a string containing the description information stored by the DFU about the specified *filename*. This description is set either through ECL watch or by using the STD.File.SetFileDescription function.

Example:

```
A := STD.File.GetFileDescription('Fred');
```

GetLogicalFileAttribute

STD.File.GetLogicalFileAttribute(*logicalfilename*, *attrname*)

<i>logicalfilename</i>	A null-terminated string containing the name of the logical file as it is known by the DFU.
<i>attrname</i>	A null-terminated string containing the name of the file attribute to return. Possible values are recordSize, recordCount, size, clusterName, directory, owner, description, ECL, partmask, numparts, name, modified, format, job, checkSum, kind, csvSeparate, csvTerminate, headerLength, footerLength, rowTag, workunit, accessed, maxRecordSize, csvQuote, blockCompressed, compressedSize, fileCrc, formatCrc, or protected. The value is case-sensitive.
Return:	GetLogicalFileAttribute returns returns a VARSTRING (null-terminated) value.

The **GetLogicalFileAttribute** function returns the value of the *attrname* for the specified *logicalfilename*.

Example:

```
IMPORT STD;
file := '~class::bmf::join::halfkeyed';

OUTPUT(STD.File.GetLogicalFileAttribute(file, 'recordSize'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'recordCount'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'size'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'clusterName'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'directory'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'numparts'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'owner'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'description'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'ECL'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'partmask'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'numparts'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'name'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'modified'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'protected'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'format'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'job'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'checkSum'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'kind'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'csvSeparate'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'csvTerminate'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'headerLength'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'footerLength'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'rowtag'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'workunit'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'accessed'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'maxRecordSize'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'csvQuote'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'blockCompressed '));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'compressedSize'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'fileCrc'));
OUTPUT(STD.File.GetLogicalFileAttribute(file, 'formatCrc'));
```

ProtectLogicalFile

STD.File.ProtectLogicalFile(*logicalfilename* [, *value*])

<i>logicalfilename</i>	A null-terminated string containing the name of the logical file as it is known by the DFU.
<i>value</i>	Optional. A boolean flag indicating whether to protect or un-protect the file. If omitted, the default is TRUE.

The **ProtectLogicalFile** function toggles protection on and off for the specified *logicalfilename*.

Example:

```
IMPORT STD;
file := '~class::bmf::join::halfkeyed';

STD.File.ProtectLogicalFile(file);           //protect
STD.File.ProtectLogicalFile(file, FALSE);    //unprotect
```

RenameLogicalFile

STD.File.RenameLogicalFile(*filename*, *newname*)

<i>filename</i>	A null-terminated string containing the current logical name of the file.
<i>newname</i>	A null-terminated string containing the new logical name for the file.

The **RenameLogicalFile** function changes the logical *filename* to the *newname*.

Example:

```
A := STD.File.RenameLogicalFile('Fred', 'Freddie');
```


SetFileDescription

STD.File.SetFileDescription(*filename* , *value*)

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>value</i>	A null-terminated string containing the description to place on the file.

The **SetFileDescription** function changes the description information stored by the DFU about the specified *filename* to the specified *value*. This description is seen either through ECL watch or by using the STD.File.GetFileDescription function.

Example:

```
A := STD.File.SetFileDescription('Fred', 'All the Freds in the world');
```

SetReadOnly

STD.File.SetReadOnly(*filename* , *flag*)

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>flag</i>	A boolean value indicating which way to set the read-only attribute of the <i>filename</i> .

The **SetReadOnly** function toggles the read-only attribute of the filename. If the *flag* is TRUE, read-only is set on.

Example:

```
A := STD.File.SetReadOnly('Fred', TRUE);  
//set read only flag on
```

VerifyFile

STD.File.VerifyFile(*file*, *usecrcs*)

<i>file</i>	A null-terminated string containing the logical name of the file.
<i>usecrcs</i>	A boolean TRUE/FALSE flag indicating that, when TRUE, compares physical CRCs of all the parts on disk. This may be slow on large files.
Return:	VerifyFile returns returns a VARSTRING value.

The **VerifyFile** function checks the system datastore (Dali) information for the *file* against the physical parts on disk and returns the following values:

OK	The file parts match the datastore information
Could not find file: <i>filename</i>	The logical <i>filename</i> was not found
Could not find part file: <i>partname</i>	The <i>partname</i> was not found
Modified time differs for: <i>partname</i>	The <i>partname</i> has a different timestamp
File size differs for: <i>partname</i>	The <i>partname</i> has a file size
File CRC differs for: <i>partname</i>	The <i>partname</i> has a different CRC

Example:

```
A := STD.File.VerifyFile('Fred1', TRUE);
```

SuperFiles

CreateSuperFile

STD.File.CreateSuperFile(*superfile* [, *sequentialparts*] [, *allowExist*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>sequentialparts</i>	Optional. A boolean value indicating whether the sub-files must be sequentially ordered. If omitted, the default is FALSE.
<i>allowExist</i>	Optional. A boolean value indicating whether to post an error if the <i>superfile</i> already exists. If TRUE, no error is posted. If omitted, the default is FALSE.
Return:	Null.

The **CreateSuperFile** function creates an empty *superfile*. This function is not included in a superfile transaction.

The *sequentialparts* parameter set to TRUE governs the unusual case where the logical numbering of sub-files must be sequential (for example, where all sub-files are already globally sorted). With *sequentialparts* FALSE (the default) the subfile parts are interleaved so the parts are found locally.

For example, if on a 4-way cluster there are 3 files (A, B, and C) then the parts are as follows:

A._1_of_4, B._1_of_4, and C._1_of_4 are on node 1

A._2_of_4, B._2_of_4, and C._2_of_4 are on node 2

A._3_of_4, B._3_of_4, and C._3_of_4 are on node 3

A._4_of_4, B._4_of_4, and C._4_of_4 are on node 4

Reading the superfile created with *sequentialparts* FALSE on Thor will read the parts in the order:

[A1,B1,C1,] [A2,B2,C2,] [A3,B3,C3,] [A4,B4,C4]

so the reads will all be local (i.e., A1,B1,C1 on node 1 etc). Setting *sequentialparts* to TRUE will read the parts in subfile order, like this:

[A1,A2,A3,] [A4,B1,B2] [,B3,B4,C1,] [C2,C3,C4]

so that the global order of A,B,C,D is maintained. However, the parts cannot all be read locally (e.g., A2 and A3 will be read on part 1). Because of this it is much less efficient to set *sequentialparts* true, and as it is unusual anyway to have files that are partitioned in order, it becomes a very unusual option to set.

Example:

```
STD.File.CreateSuperFile('~CLASS::RT::IN::SF1',,1);  
//This is the same but uses named parameter  
STD.File.CreateSuperFile('~CLASS::RT::IN::SF1',allowExist := 1);
```

SuperFileExists

STD.File.SuperFileExists(*filename*)

<i>filename</i>	A null-terminated string containing the logical name of the superfile.
Return:	SuperFileExists returns a BOOLEAN value.

The **SuperFileExists** function returns TRUE if the specified *filename* is present in the Distributed File Utility (DFU) and is a SuperFile. It returns FALSE if the *filename* does exist but it is not a SuperFile (in other words, it is a normal DATASET. Use the STD.File.FileExists function to detect their presence or absence).

This function is not included in a superfile transaction.

Example:

```
A := STD.File.SuperFileExists('~CLASS::RT::IN::SF1');
```

See Also: FileExists

DeleteSuperFile

STD.File.DeleteSuperFile(*superfile* [, *subdeleteflag*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subdeleteflag</i>	A boolean value indicating whether to delete the sub-files. If omitted, the default is FALSE. This option should not be used if the superfile contains any foreign file or foreign superfile.
Return:	Null.

The **DeleteSuperFile** function deletes the *superfile*.

This function is not included in a superfile transaction.

Example:

```
STD.File.DeleteSuperFile('~CLASS::RT::IN::SF1');
```

GetSuperFileSubCount

STD.File.GetSuperFileSubCount(*superfile*)

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
Return:	GetSuperFileSubCount returns an INTEGER4 value.

The **GetSuperFileSubCount** function returns the number of sub-files comprising the *superfile*.

This function is not included in a superfile transaction.

Example:

```
A := STD.File.GetSuperFileSubCount('~CLASS::RT::IN::SF1');
```


GetSuperFileSubName

STD.File.GetSuperFileSubName(*superfile*, *subfile* [, *absolutePath*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subfile</i>	An integer in the range of one (1) to the total number of sub-files in the <i>superfile</i> specifying the ordinal position of the sub-file whose name to return.
<i>absolutePath</i>	Optional. A boolean TRUE/FALSE to indicate whether to prepend a tilde (~) to the resulting foreign logical file name. If omitted, the default is FALSE.
Return:	GetSuperFileSubName returns a VARSTRING value.

The **GetSuperFileSubName** function returns the logical name of the specified *subfile* in the *superfile*.

This function is not included in a superfile transaction.

Example:

```
A := STD.File.GetSuperFileSubName('~CLASS::RT::IN::SF1', 1);
    //get name of first sub-file
//this example gets the name of the first sub-file in
// a foreign superfile
sf := '~thor_data400::BASE::Business_Header';
sub := STD.File.GetSuperFileSubName( STD.File.ForeignLogicalFileName (sf,
    '10.150.29.161',
    TRUE),
    1, TRUE);
OUTPUT(STD.File.ForeignLogicalFileName(sub, ''));
```

LogicalFileSuperOwners

STD.File.LogicalFileSuperOwners(*filename*)

<i>filename</i>	A null-terminated string containing the logical name of the file.
Return:	LogicalFileSuperOwners returns a dataset in the following format:

```
EXPORT FsLogicalFileNameRecord := RECORD
  STRING name;
END;
```

The **LogicalFileSuperOwners** function returns a list of the logical filenames of all the SuperFiles that contain the *filename* as a sub-file.

This function is not included in a superfile transaction.

Example:

```
OUTPUT(STD.File.LogicalFileSuperowners('~CLASS::RT::SF::Daily1'));
//returns all SuperFiles that "own" the Daily1 file
```

LogicalFileSuperSubList

STD.File.LogicalFileSuperSubList()

Return: LogicalFileSuperSubList returns a dataset in the following format:

```
EXPORT FsLogicalSuperSubRecord := RECORD
  STRING supname{MAXLENGTH(255)};
  STRING subname{MAXLENGTH(255)};
END;
```

The **LogicalFileSuperSubList** function returns a list of the logical filenames of all the SuperFiles and their component sub-files.

This function is not included in a superfile transaction.

Example:

```
OUTPUT(STD.File.LogicalFileSuperSubList());
//returns all SuperFiles and their sub-files
```

SuperFileContents

STD.File.SuperFileContents(*filename* [, *recurse*])

<i>filename</i>	A null-terminated string containing the logical name of the SuperFile.
<i>recurse</i>	A boolean flag indicating whether to expand nested SuperFiles within the filename so that only logical files are returned. If omitted, the default is FALSE.
Return:	SuperFileContents returns a dataset in the following format:

```
EXPORT FsLogicalFileNameRecord := RECORD
  STRING name;
END;
```

The **SuperFileContents** function returns a list of the logical filenames of all the sub-files in the *filename*.

This function is not included in a superfile transaction.

Example:

```
OUTPUT(STD.File.SuperFileContents('~CLASS::RT::SF::Daily'));
//returns all files in the SuperFile
```

FindSuperFileSubName

STD.File.FindSuperFileSubName(*superfile*, *subfile*)

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subfile</i>	A null-terminated string containing the logical name of the sub-file.
Return:	FindSuperFileSubName returns an INTEGER4 value.

The **FindSuperFileSubName** function returns the ordinal position of the specified *subfile* in the *superfile*.

This function is not included in a superfile transaction.

Example:

```
A := STD.File.GetSuperFileSubName('~CLASS::RT::IN::SF1', 'Sue'); //get position of sub-file Sue
```

StartSuperFileTransaction

STD.File.StartSuperFileTransaction()

Return: Null.

The **StartSuperFileTransaction** function begins a transaction frame for superfile maintenance. The transaction frame is terminated by calling the **FinishSuperFileTransaction** function. Within the transaction frame, multiple superfiles may be maintained by adding, removing, clearing, swapping, and replacing sub-files.

The **FinishSuperFileTransaction** function does an automatic rollback of the transaction if any error or failure occurs during the transaction frame. If no error occurs, then the commit or rollback of the transaction is controlled by the *rollback* parameter to the **FinishSuperFileTransaction** function.

Example:

```
STD.File.StartSuperFileTransaction();
```

AddSuperFile

STD.File.AddSuperFile(*superfile*, *subfile* [, *atpos*] [, *addcontents*] [, *strict*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subfile</i>	A null-terminated string containing the logical name of the sub-file. This may be another superfile.
<i>atpos</i>	An integer specifying the position of the <i>subfile</i> in the <i>superfile</i> . If omitted, the default is zero (0), which places the <i>subfile</i> at the end of the <i>superfile</i> .
<i>addcontents</i>	A boolean flag that, if set to TRUE, specifies the <i>subfile</i> is also a superfile and the contents of that superfile are added to the superfile rather than its reference. If omitted, the default is to add by reference (<i>addcontents</i> := FALSE).
<i>strict</i>	A boolean flag specifying, in the case of a <i>subfile</i> that is itself a superfile, whether to check for the existence of the superfile and raise an error if it does not. Also, if <i>addcontents</i> is set to TRUE, it will ensure the <i>subfile</i> that is itself a superfile is not empty. If omitted, the default is false.
Return:	Null.

The **AddSuperFile** function adds the *subfile* to the list of files comprising the *superfile*. All *subfiles* in the *superfile* must have exactly the same structure type and format.

This function may be included in a superfile transaction, but is not required to be.

Example:

```
IMPORT STD;
SEQUENTIAL(
  STD.File.StartSuperFileTransaction(),
  STD.File.AddSuperFile('MySuperFile1','MySubFile1'),
  STD.File.AddSuperFile('MySuperFile1','MySubFile2'),
  STD.File.AddSuperFile('MySuperFile2','MySuperFile1'),
  STD.File.AddSuperFile('MySuperFile3','MySuperFile1',addcontents := true),
  STD.File.FinishSuperFileTransaction()
);

// MySuperFile1 contains { MySubFile1, MySubFile2 }
// MySuperFile2 contains { MySuperFile1 }
// MySuperFile3 contains { MySubFile1, MySubFile2 }
```

RemoveSuperFile

STD.File.RemoveSuperFile(*superfile*, *subfile* [, *delete*] [, *removecontents*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subfile</i>	A null-terminated string containing the logical name of the sub-file. This may be another superfile or a foreign file or superfile.
<i>delete</i>	A boolean flag specifying whether to delete the <i>subfile</i> from disk or just remove it from the <i>superfile</i> list of files. If omitted, the default is to just remove it from the <i>superfile</i> list of files. This option should not be used if the subfile is a foreign file or foreign superfile.
<i>removecontents</i>	A boolean flag specifying whether the contents of a <i>subfile</i> that is itself a superfile are recursively removed.
Return:	Null.

The **RemoveSuperFile** function removes the *subfile* from the list of files comprising the *superfile*.

This function may be included in a superfile transaction.

Example:

```
SEQUENTIAL(  
  STD.File.StartSuperFileTransaction(),  
  STD.File.RemoveSuperFile('MySuperFile', 'MySubFile'),  
  STD.File.FinishSuperFileTransaction()  
);
```


ClearSuperFile

STD.File.ClearSuperFile(*superfile*, [, *delete*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>delete</i>	A boolean flag specifying whether to delete the sub-files from disk or just remove them from the <i>superfile</i> list of files. If omitted, the default is to just remove them from the <i>superfile</i> list of files.
Return:	Null.

The **ClearSuperFile** function removes all sub-files from the list of files comprising the *superfile*.

This function may be included in a superfile transaction.

Example:

```
SEQUENTIAL(  
  STD.File.StartSuperFileTransaction(),  
  STD.File.ClearSuperFile('MySuperFile'),  
  STD.File.FinishSuperFileTransaction()  
);
```

RemoveOwnedSubFiles

STD.File.RemoveOwnedSubFiles(*superfile* [, *delete*])

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>delete</i>	A boolean flag specifying to delete the sub-files from disk when TRUE or just remove them from the <i>superfile</i> list of files. If omitted, the default is to just remove them from the <i>superfile</i> list of files.
Return:	Null.

The **RemoveOwnedSubFiles** function removes all owned sub-files from the specified superfile. These are only removed if they are solely owned by the superfile. If a subfile is co-owned, (i.e., a member of any other superfile), then the removal is ignored.

This function may be included in a superfile transaction, unless the delete Flag is TRUE.

Example:

```
SEQUENTIAL(  
  STD.File.StartSuperFileTransaction(),  
  STD.File.RemoveOwnedSubFiles('MySuperFile'),  
  STD.File.FinishSuperFileTransaction()  
);
```

SwapSuperFile

STD.File.SwapSuperFile(*superfile1*, *superfile2*)

<i>superfile1</i>	A null-terminated string containing the logical name of the superfile.
<i>superfile2</i>	A null-terminated string containing the logical name of the superfile.
Return:	Null.

The **SwapSuperFile** function moves all sub-files from *superfile1* to *superfile2* and vice versa.

This function may be included in a superfile transaction.

Example:

```
SEQUENTIAL(  
  STD.File.StartSuperFileTransaction(),  
  STD.File.SwapSuperFile('MySuperFile','YourSuperFile'),  
  STD.File.FinishSuperFileTransaction()  
);
```

ReplaceSuperFile

STD.File.ReplaceSuperFile(*superfile*, *subfile1* , *subfile2*)

<i>superfile</i>	A null-terminated string containing the logical name of the superfile.
<i>subfile1</i>	A null-terminated string containing the logical name of the sub-file. This may be another super-file.
<i>subfile2</i>	A null-terminated string containing the logical name of the sub-file. This may be another super-file.
Return:	Null.

The **ReplaceSuperFile** function removes the *subfile1* from the list of files comprising the *superfile* and replaces it with *subfile2*.

This function may be included in a superfile transaction.

Example:

```
SEQUENTIAL(  
  STD.File.StartSuperFileTransaction(),  
  STD.File.ReplaceSuperFile('MySuperFile',  
    'MyOldSubFile',  
    'MyNewSubFile'),  
  STD.File.FinishSuperFileTransaction()  
);
```

PromoteSuperFileList

STD.File.PromoteSuperFileList(*supernames* [, *addhead*] [, *deltail*] [, *createjustone*] [, *reverse*])

oldlist := **STD.File.fPromoteSuperFileList**(*supernames* [, *addhead*] [, *deltail*] [, *createjustone*] [, *reverse*]);

<i>supernames</i>	A set of null-terminated strings containing the logical names of the superfiles to act on. Any that don't exist will be created. The contents of each superfile will be moved to the next in the list (NB -- each superfile must contain different sub-files).
<i>addhead</i>	Optional. A null-terminated string containing a comma-delimited list of logical file names to add to the first <i>superfile</i> after the promotion process is complete.
<i>deltail</i>	Optional. A boolean value specifying whether to physically delete the contents moved out of the last superfile. If omitted, the default is FALSE.
<i>createjustone</i>	Optional. A boolean value specifying whether to only create a single superfile (truncate the list at the first non-existent superfile). If omitted, the default is FALSE.
<i>reverse</i>	Optional. A boolean value specifying whether to reverse the order of processing the <i>supernames</i> list, effectively "demoting" instead of "promoting" the sub-files. If omitted, the default is FALSE.
<i>oldlist</i>	The name of the attribute that receives the returned string containing the list of the previous subfile contents of the emptied superfile.
Return:	PromoteSupeFileList returns Null; fPromoteSupeFileList returns a string.

The **PromoteSuperFileList** function moves the subfiles from the first entry in the list of *supernames* to the next in the list, subsequently repeating the process through the list of *supernames*.

This function does not use superfile transactions, it is an atomic operation.

Example:

```
STD.File.PromoteSuperFileList(['Super1', 'Super2', 'Super3'],  
                              'NewSub1');  
//Moves what was in Super1 to Super2,  
// what was in Super2 to Super3, replacing what was in Super3,  
// and putting NewSub1 in Super1
```

FinishSuperFileTransaction

STD.File.FinishSuperFileTransaction([*rollback*])

<i>rollback</i>	Optional. A boolean flag that indicates whether to commit (FALSE) or roll back (TRUE) the transaction. If omitted, the default is FALSE.
Return:	Null.

The **FinishSuperFileTransaction** function terminates a superfile maintenance transaction frame. If the *rollback* flag is FALSE, the transaction is committed atomically and the transaction frame closes. Otherwise, the transaction is rolled back and the transaction frame closes.

Example:

```
STD.File.FinishSuperFileTransaction();
```

External Files

ExternalLogicalFileName

STD.File.ExternalLogicalFileName(*machineIP*, *filename*)

<i>machineIP</i>	A null-terminated string containing the IP address of the remote machine.
<i>filename</i>	A null-terminated string containing the path/name of the file.
Return:	ExternalLogicalFileName returns returns a VARSTRING (null-terminated) value.

The **ExternalLogicalFileName** function returns an appropriately encoded external logical file name that can be used to directly read a file from any node that is running the dafilesrv utility (typically a landing zone). It handles upper case characters by escaping those characters in the return string.

Example:

```
IP    := '10.150.254.6';
file  := '/c$/training/import/AdvancedECL/people';
DS1   := DATASET(STD.File.ExternalLogicalFileName(IP,file),
                 Training_Advanced.Layout_PeopleFile, FLAT);
OUTPUT(STD.File.ExternalLogicalFileName(IP,file));
//returns:
//~file::10.150.254.6::c$::training::import::^advanced^e^c^l::people
OUTPUT(DS1);
//returns records from the external file
```


MoveExternalFile

STD.File.MoveExternalFile(*location*, *frompath*, *topath*)

<i>location</i>	A null-terminated string containing the IP address of the remote machine.
<i>frompath</i>	A null-terminated string containing the path/name of the file to move.
<i>topath</i>	A null-terminated string containing the path/name of the target file.

The **MoveExternalFile** function moves the single physical file specified by the *frompath* to the *topath*. Both *frompath* and *topath* are on the same remote machine, identified by the *location*. The dafileserv utility program must be running on the *location* machine.

Example:

```
IP      := '10.150.254.6';
infile  := '/c$/training/import/AdvancedECL/people';
outfile := '/c$/training/import/DFUtest/people';
STD.File.MoveExternalFile(IP,infile,outfile);
```

DeleteExternalFile

STD.File.DeleteExternalFile(*location*, *path*)

<i>location</i>	A null-terminated string containing the IP address of the remote machine.
<i>path</i>	A null-terminated string containing the path/name of the file to remove.

The **DeleteExternalFile** function removes the single physical file specified by the *path* from the *location*. The *dafile-serv* utility program must be running on the *location* machine.

Example:

```
IP    := '10.150.254.6';  
infile := '/c$/training/import/AdvancedECL/people';  
STD.File.DeleteExternalFile(IP,infile);
```

CreateExternalDirectory

STD.File.CreateExternalDirectory(*location*, *path*)

<i>location</i>	A null-terminated string containing the IP address of the remote machine.
<i>path</i>	A null-terminated string containing the directory path to create.

The **CreateExternalDirectory** function creates the *path* on the *location* (if it does not already exist). The *dafileserv* utility program must be running on the *location* machine.

Example:

```
IP    := '10.150.254.6';  
path := '/c$/training/import/NewDir';  
STD.File.CreateExternalDirectory(IP,path);
```

RemoteDirectory

STD.File.RemoteDirectory(*machineIP*, *directory* [, *mask*] [, *includesubs*])

<i>machineIP</i>	A null-terminated string containing the IP address of the remote machine.
<i>directory</i>	A null-terminated string containing the path to the directory to read. This must be in the appropriate format for the operating system running on the remote machine.
<i>mask</i>	Optional. A null-terminated string containing the filemask specifying which files to include in the result. If omitted, the default is '*' (all files).
<i>includesubdir</i>	Optional. A boolean flag indicating whether to include files from sub-directories under the <i>directory</i> . If omitted, the default is FALSE.
Return:	RemoteDirectory returns a dataset in the following format:

```
EXPORT FsFilenameRecord := RECORD
  STRING name;          //filename
  UNSIGNED8 size;       //filesize
  STRING19 modified;    //date-time stamp
END;
```

The **RemoteDirectory** function returns a list of files as a dataset in the format listed above from the specified *machineIP* and *directory*. If *includesubdir* is set to TRUE, then the name field contains the relative path to the file from the specified *directory*.

Example:

```
OUTPUT(STD.File.RemoteDirectory('edata12', '\in', '*.d00'));
OUTPUT(STD.File.RemoteDirectory('10.150.254.6',
  '/c$/training', TRUE));
```

File Browsing

SetColumnMapping

STD.File.SetColumnMapping(*file*, *mapping*);

<i>file</i>	A null-terminated string containing the logical filename.
<i>mapping</i>	A null-terminated string containing a comma-delimited list of field mappings.

The **SetColumnMapping** function defines how the data in the fields of the *file* must be transformed between the actual data storage format and the input format used to query that data.

The format for each field in the *mapping* list is:

<field>{set(<transform>(args),...),get(<transform>,...),displayname(<name>)}

<field>	The name of the field in the file.
set	Optional. Specifies the transforms applied to the values supplied by the user to convert them to values in the file.
<transform>	Optional. The name of a function to apply to the value. This is typically the name of a plugin function. The value being converted is always provided as the first parameter to the function, but extra parameters can be specified in brackets after the transform name (similar to SALT hygiene).
get	Optional. Specifies the transforms applied to the values in the file to convert them to the formatted values as they are understood by the user.
displayname	Optional. Allows a different <i>name</i> to be associated with the field than the user would naturally understand.

Note that you may mix unicode and string functions, as the system automatically converts the parameters to the appropriate types expected for the functions.

Example:

```
// A file where the firstname(string) and lastname(unicode) are
//always upper-cased:
// There is no need for a displayname since it isn't really a
// different field as far as the user is concerned, and there is
// obviously no get transformations.
  firstname{set(stringlib.StringToUpperCase)},
             surname{set(unicodeLib.UnicodeToUpperCase)}
// A name translated using a phonetic key
// it is worth specifying a display name here, because it will make
// more sense to the user, and the user may want to enter either the
// translated or untranslated names.
  dph_lname{set(metaphonelib.DMetaPhone1),
             displayname(lname)}
// A file where a name is converted to a token using the namelib
// functions. (I don't think we have an example of this)
// (one of the few situations where a get() attribute is useful)
  fnametoken{set(namelib.nameToToken),
              get(namelib.tokenToName),
              displayname(fname)}
// upper case, and only include digits and alphabetic.
  searchname{set(stringlib.StringToUpperCase,
                 stringlib.StringFilter(
                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'))}
// A file with a field that that needs to remove accents and then
// uppercase:
  lastname{set(unicodeLib.CleanAccents,stringLib.StringToUpperCase)}
```

GetColumnMapping

result := **STD.File.GetColumnMapping**(*file*);

<i>file</i>	A null-terminated string containing the logical filename.
Return:	GetColumnMapping returns a null-terminated string containing the comma-delimited list of field mappings for the <i>file</i> .

The **GetColumnMapping** function returns the field mappings for the *file*, in the same format specified for the SetColumnMapping function.

Example:

```
Maps := STD.File.GetColumnMapping('Thor::in::SomeFile');
```

AddFileRelationship

STD.File.AddFileRelationship(*primary*, *secondary*, *primaryfields*, *secondaryfields*, [*relationship*], *cardinality*, *payload* [, *description*]);

<i>primary</i>	A null-terminated string containing the logical filename of the primary file.
<i>secondary</i>	A null-terminated string containing the logical filename of the secondary file.
<i>primaryfields</i>	A null-terminated string containing the name of the primary key field for the <i>primary</i> file. The value “__fileposition__” indicates the <i>secondary</i> is an INDEX that must use FETCH to access non-keyed fields.
<i>secondaryfields</i>	A null-terminated string containing the name of the foreign key field relating to the <i>primary</i> file.
<i>relationship</i>	A null-terminated string containing either “link” or “view” indicating the type of relationship between the <i>primary</i> and <i>secondary</i> files. If omitted, the default is “link.”
<i>cardinality</i>	A null-terminated string containing the kind of relationship between the <i>primary</i> and <i>secondary</i> files. The format is X:Y where X indicates the <i>primary</i> and Y indicates the <i>secondary</i> . Valid values for X and Y are “1” or ‘M’.
<i>payload</i>	A BOOLEAN value indicating whether the <i>primary</i> or <i>secondary</i> are payload INDEXes.
<i>description</i>	A null-terminated string containing the relationship description.

The **AddFileRelationship** function defines the relationship between two files. These may be DATASETs or INDEXes. Each record in the *primary* file should be uniquely defined by the *primaryfields* (ideally), preferably efficiently.

The *primaryfields* and *secondaryfields* parameters can have the same format as the column mappings for a file (see the SetColumnMappings function documentation) , although they will often be just a list of fields.

They are currently used in two different ways:

First, the roxie browser needs a way of determining which indexes are built from which files. A “view” relationship should be added each time an index is built from a file, like this:

```
STD.File.AddFileRelationship(DG_FlatFileName, DG_IndexFileName,  
                             '', '', 'view', '1:1', false);
```

To implement the roxie browser there is no need to define the *primaryfields* or *secondaryfields*, so passing blank strings is recommended.

Second, the browser needs a way of finding all the original information from the file from an index.

This stage depends on the nature of the index:

- If the index contains all the relevant data from the original file you don't need to do anything.
- If the index uses a fileposition field to FETCH extra data from the original file then add a relationship between the original file and the index, using a special value of __fileposition__ to indicate the record is retrieved using a FETCH.

```
STD.File.AddFileRelationship('fetch_file',  
                             'new_index',  
                             '__fileposition__',  
                             'index_filepos_field',  
                             'link',  
                             '1:1',  
                             true);
```


The original file is the primary, since the rows are uniquely identified by the fileposition (also true of the index), and the retrieval is efficient.

c) If the index uses a payload field which needs to be looked up in another index to provide the information, then you need to define a relationship between the new index and the index that provides the extra information. The index providing the extra information is the primary.

```
STD.File.AddFileRelationship('related_index',  
                             'new_index',  
                             'related_key_fields',  
                             'index_filepos_field',  
                             'link',  
                             '1:M',  
                             true);
```

The *payload* flag is there so that the roxie browser can distinguish this link from a more general relationship between two files.

You should ensure any super-file names are expanded if the relation is defined between the particular sub-files.

While going through all the attributes it may be worth examining whether it makes sense to add relationships for any other combinations of files. It won't have any immediate beneficial effect, but would once we add an ER diagram to the system. A couple of examples may help illustrate the syntax.

For a typical example, datasets with a household and person file, the following defines a relationship linking by household id (hhid):

```
STD.File.AddFileRelationship('HHFile','PersonFile', 'hhid','hhid', 'link', '1:M', false);
```

Here's a more hypothetical example—a file query with firstname, lastname related to an index with phonetic names you might have:

```
STD.File.AddFileRelationship('names', 'inquiries', 'plastrname{set(phonetic)},  
                             pfirstname{set(phonetic)}',  
                             'lastname{set(fail)},firstname{set(fail)}', 'link', '1:M', false);
```

Note, the fail mapping indicates that you can use the phonetic mapping from inquiries to names, but there is no way of mapping from names to inquiries. There could equally be get(fail) attributes on the index fields.

Example:

```
Maps := STD.File.GetColumnMapping('Thor::in::SomeFile');
```

FileRelationshipList

STD.File.FileRelationshipList(*primary*, *secondary* [, *primaryfields*] [, *secondaryfields*] [, *relationship*]);

<i>primary</i>	A null-terminated string containing the logical filename of the primary file.
<i>secondary</i>	A null-terminated string containing the logical filename of the secondary file.
<i>primaryfields</i>	A null-terminated string containing the name of the primary key field for the <i>primary</i> file. The value “__fileposition__” indicates the <i>secondary</i> is an INDEX that must use FETCH to access non-keyed fields. If omitted, the default is an empty string.
<i>secondaryfields</i>	A null-terminated string containing the name of the foreign key field relating to the <i>primary</i> file. If omitted, the default is an empty string.
<i>relationship</i>	A null-terminated string containing either “link” or “view” indicating the type of relationship between the <i>primary</i> and <i>secondary</i> files. If omitted, the default is “link.”
Return:	FileRelationshipList returns a dataset in the FsFileRelationshipRecord format.

The **FileRelationshipList** function returns a list file relationships between the *primary* and *secondary* files. The return records are structured in the FsFileRelationshipRecord format:

```
EXPORT FsFileRelationshipRecord := RECORD
  STRING primaryfile {MAXLENGTH(1023)};
  STRING secondaryfile {MAXLENGTH(1023)};
  STRING primaryflds {MAXLENGTH(1023)};
  STRING secondaryflds {MAXLENGTH(1023)};
  STRING kind {MAXLENGTH(16)};
  STRING cardinality {MAXLENGTH(16)};
  BOOLEAN payload;
  STRING description {MAXLENGTH(1023)};
END;
```

Example:

```
OUTPUT(STD.File.FileRelationshipList('names', 'inquiries'));
```

See Also: AddFileRelationship

RemoveFileRelationship

STD.File.RemoveFileRelationship(*primary*, *secondary*, [,*primaryfields*] [,*secondaryfields*] [, *relationship*]);

<i>primary</i>	A null-terminated string containing the logical filename of the primary file.
<i>secondary</i>	A null-terminated string containing the logical filename of the secondary file.
<i>primaryfields</i>	A null-terminated string containing the name of the primary key field for the <i>primary</i> file. The value “__fileposition__” indicates the <i>secondary</i> is an INDEX that must use FETCH to access non-keyed fields. If omitted, the default is an empty string.
<i>secondaryfields</i>	A null-terminated string containing the name of the foreign key field relating to the <i>primary</i> file. If omitted, the default is an empty string.
<i>relationship</i>	A null-terminated string containing either “link” or “view” indicating the type of relationship between the <i>primary</i> and <i>secondary</i> files. If omitted, the default is “link.”

The **RemoveFileRelationship** function removes a file relationship between the *primary* and *secondary* files.

Example:

```
STD.File.RemoveFileRelationship('names', 'inquiries');
```

See Also: AddFileRelationship

File Movement

DfuPlusExec

STD.File.DfuPlusExec(*commandline*])

<i>commandline</i>	A null-terminated string containing the DFUPlus.exe command line to execute. The valid arguments are documented in the Client Tools manual, in the section describing the DfuPlus.exe program.
--------------------	--

The **DfuPlusExec** action executes the specified *commandline* just as the DfuPlus.exe program would do. This simply allows you to have all the functionality of the DfuPlus.exe program available within your ECL code.

Example:

```
IMPORT STD;

serv := 'server=http://10.150.50.12:8010 ';
user := 'username=rlor ';
pswd := 'password=password ';
over := 'overwrite=1 ';
repl := 'replicate=1 ';

action := 'action=despray ';
dstip := 'dstip=10.150.50.14 ';
dstfile := 'dstfile=c:\\import\\despray\\timezones.txt ';
srcname := 'srcname=RTTEMP::timezones.txt ';

cmd := serv + user + pswd + over + repl + action + dstip + dstfile + srcname;
STD.File.DfuPlusExec(cmd);
```

AbortDfuWorkunit

STD.File.AbortDfuWorkunit(*dfuwuid* [,*espserverIPport*])

<i>dfuwuid</i>	A null-terminated string containing the DFU workunit ID (DFUWUID) for the job to abort. This value is returned by the “leading-f” versions of the Copy, SprayFixed, SprayVariable, SprayXML, and Despray FileServices functions.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.

The **AbortDfuWorkunit** function aborts the specified DFU workunit. Typically that workunit will have been launched with its *timeout* parameter set to zero (0).

Example:

```
STD.File.AbortDfuWorkunit('D20051108-143758');
```

Copy

STD.File.Copy(*logicalname*, *destinationGroup*, *destinationLogicalname*, [*,sourceDali*] [*,timeout*] [*,espserviceIPport*] [*,maxConnections*] [*,allowoverwrite*] [*,replicate*] [*,asSuperfile*] [*,compress*] [*,forcePush*] [*,transferBufferSize*] [*,preserveCompression*]);

dfuwuid := **STD.File.fCopy**(*logicalname*, *destinationGroup*, *destinationLogicalname*, [*,sourceDali*] [*,timeout*] [*,espserviceIPport*] [*,maxConnections*] [*,allowoverwrite*] [*,replicate*] [*,asSuperfile*] [*,compress*] [*,forcePush*] [*,transferBufferSize*] [*,preserveCompression*]);

<i>logicalname</i>	A null-terminated string containing the logical name of the file.
<i>destinationGroup</i>	A null-terminated string containing the destination cluster for the file.
<i>destinationLogicalname</i>	A null-terminated string containing the new logical name of the file.
<i>sourceDali</i>	Optional. A null-terminated string containing the IP and Port of the Dali containing the file to copy. If omitted, the default is an intra-Dali copy.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserviceIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/ FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is one (1).
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>replicate</i>	Optional. A boolean TRUE or FALSE flag indicating whether to automatically replicate the new file. If omitted, the default is FALSE.
<i>asSuperfile</i>	Optional. A boolean TRUE or FALSE flag indicating whether to treat the file as a superfile. If omitted, the default is FALSE. If TRUE and the file to copy is a superfile, then the operation creates a superfile on the target, creating subfiles as needed while overwriting only those already existing subfiles whose content has changed. If FALSE and the file to copy is a superfile, then the operation consolidates all the superfile content into a single logical file on the target, not a superfile.
<i>compress</i>	Optional. A boolean TRUE or FALSE flag indicating whether to LZW compress the new file. If omitted, the default is FALSE.
<i>forcePush</i>	Optional. A boolean TRUE or FALSE flag indicating whether to execute the copy process on the source nodes and push to the targets instead of executing on the targets and pulling from the source. If omitted, the default is FALSE.
<i>transferBufferSize</i>	Optional. An integer value to override the DFU Server's buffer size value (default is 64k)
<i>preserveCompression</i>	Optional. A boolean TRUE or FALSE flag indicating whether to preserve the compression of the old file when copying. If omitted, the default is TRUE.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	Copy returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **Copy** function takes a logical file and copies it to another logical file. This may be done within the same cluster, or to another cluster, or to a cluster in a completely separate Dali.

Example:

```
STD.File.Copy('OUT::MyFile', '400way', 'OUT::MyNewFile');
```


DeSpray

STD.File.DeSpray(*logicalname*, *destinationIP* , *destinationpath* , [*timeout*] , [*espserverIPport*] , [*maxConnections*] , [*allowoverwrite*])

dfuwuid := **STD.File.fDeSpray**(*logicalname*, *destinationIP* , *destinationpath* , [*timeout*] , [*espserverIPport*] , [*maxConnections*] , [*allowoverwrite*]);

<i>logicalname</i>	A null-terminated string containing the logical name of the file.
<i>destinationIP</i>	A null-terminated string containing the destination IP address of the file.
<i>destinationpath</i>	A null-terminated string containing the path and name of the file.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is one (1).
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	fDeSpray returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **DeSpray** function takes a logical file and desprays it (combines all parts on each supercomputer node into a single physical file) to the landing zone.

Example:

```
STD.File.DeSpray( 'OUT:MyFile',  
  '10.150.50.14',  
  'c:\\OutputData\\MyFile.txt',  
  -1,  
  'http://10.150.50.12:8010/FileSpray' );
```

RemotePull

STD.File.RemotePull(*remoteURL*, *sourceLogicalname*, *destinationGroup* , *destinationLogicalname*, [*,timeout*] [*,maxConnections*] [*,allowoverwrite*] [*,replicate*] [*,asSuperfile*] [*,forcePush*] [*,transferBufferSize*] [*,wrap*] [*,compress*])

dfuwuid := **STD.File.RemotePull**(*remoteURL*, *sourceLogicalname*, *destinationGroup* , *destinationLogicalname*, [*,timeout*] [*,maxConnections*] [*,allowoverwrite*] [*,replicate*] [*,asSuperfile*] [*,forcePush*] [*,transferBufferSize*] [*,wrap*] [*,compress*]);

<i>remoteURL</i>	A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the remote ESP server program. This is usually the same IP and port as its ECL Watch, with “/FileSpray” appended.
<i>sourceLogicalname</i>	A null-terminated string containing the local logical name of the file.
<i>destinationGroup</i>	A null-terminated string containing the name of the destination cluster.
<i>destinationLogicalname</i>	A null-terminated string containing the logical name to give the file on the remote cluster (this must be completely specified, including the domain).
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is -1, which indicates the system chooses a suitable default based on the size of the cluster.
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>replicate</i>	Optional. A boolean TRUE or FALSE flag indicating whether to automatically replicate the new file. If omitted, the default is FALSE.
<i>asSuperfile</i>	Optional. A boolean TRUE or FALSE flag indicating whether to treat the file as a superfile. If omitted, the default is FALSE. If TRUE and the file to copy is a superfile, then the operation creates a superfile on the target, creating subfiles as needed while overwriting only those already existing subfiles whose content has changed. If FALSE and the file to copy is a superfile, then the operation consolidates all the superfile content into a single logical file on the target, not a superfile.
<i>forcePush</i>	Optional. A boolean TRUE or FALSE flag indicating whether to execute the copy process on the source nodes and push to the targets instead of executing on the targets and pulling from the source. If omitted, the default is FALSE.
<i>transferBufferSize</i>	Optional. An integer specifying the size in bytes of the transfer buffer. Sometimes using larger values can speed the process. If omitted, a default buffer size of 64K is used.
<i>wrap</i>	Optional. A boolean TRUE or FALSE flag indicating whether to automatically wrap the file parts when copying to smaller sized clusters. For example, copying from a 6-node cluster to a 3-node cluster, two file parts will end up on each node; the difference is whether node 1 gets parts 1 and 2 or parts 1 and 4. If omitted, the default is FALSE.
<i>compress</i>	Optional. A boolean TRUE or FALSE flag indicating whether to automatically LZW compress the new file. If omitted, the default is FALSE.
<i>dfuwuid</i>	The definition name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	fRemotePull returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **RemotePull** function executes on the *remoteURL*, copying the *sourceLogicalName* from the local environment that instantiated the operation to the remote environment's *destinationGroup* cluster, giving it the *destinationLogicalName*. This is very similar to using the STD.File.Copy function and specifying its *espserverIPport* parameter. Since the DFU workunit executes on the remote DFU server, the user name authentication must be the same on both systems, and the user must have rights to copy files on both systems.

Example:

```
STD.File.RemotePull('http://10.150.50.14:8010/FileSpray',  
    '~THOR::LOCAL::MyFile',  
    'RemoteThor',  
    '~REMOTETHOR::LOCAL::MyFile');
```

Replicate

STD.File.Replicate (*filename* [, *timeout*] [, *espserverIPport*])

dfuwuid := **STD.File.fReplicate**(*filename* [, *timeout*] [, *espserverIPport*]);

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/ FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.

The **Replicate** function copies the individual parts of the *filename* to the mirror disks for the cluster. Typically, this means that the file part on one node's C drive is copied to its neighbors D drive.

Example:

```
A := STD.File.Replicate('Fred');
```

SprayFixed

STD.File.SprayFixed(*sourceIP* , *sourcepath*, *recordsize*, *destinationgroup*, *destinationlogicalname* , [*timeout*] , [*espserviceIPport*] , [*maxConnections*] , [*allowoverwrite*] , [*replicate*] , [*compress*])

dfuwuid := **STD.File.fSprayFixed**(*sourceIP* , *sourcepath*, *recordsize*, *destinationgroup*, *destinationlogicalname* , [*timeout*] , [*espserviceIPport*] , [*maxConnections*] , [*allowoverwrite*] , [*replicate*] , [*compress*]);

<i>sourceIP</i>	A null-terminated string containing the IP address of the file.
<i>sourcepath</i>	A null-terminated string containing the path and name of the file.
<i>recordsize</i>	An integer containing the size of the records in the file.
<i>destinationgroup</i>	A null-terminated string containing the name of the specific supercomputer within the target cluster.
<i>destinationlogicalname</i>	A null-terminated string containing the logical name of the file.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserviceIPport</i>	A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is one (1).
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>replicate</i>	Optional. A boolean TRUE or FALSE flag indicating whether to replicate the new file. If omitted, the default is FALSE.
<i>compress</i>	Optional. A boolean TRUE or FALSE flag indicating whether to compress the new file. If omitted, the default is FALSE.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	fSprayFixed returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **SprayFixed** function takes fixed-format file from the landing zone and distributes it across the nodes of the destination supercomputer.

Example:

```
STD.File.SprayFixed('10.150.50.14', 'c:\\InputData\\MyFile.txt',  
255, '400way', 'IN::MyFile', -1,  
'http://10.150.50.12:8010/FileSpray');
```

SprayVariable

STD.File.SprayVariable(*sourceIP* , *sourcepath* , [*maxrecordsize*] , [*srcCSVseparator*] , [*srcCSVterminator*] , [*srcCSVquote*] , *destinationgroup* , *destinationlogicalname* , [*timeout*] , [*espserverIPport*] , [*maxConnections*] , [*allowoverwrite*] , [*replicate*] , [*compress*] , [*sourceCsvEscape*])

dfuwuid := **STD.File.fSprayVariable**(*sourceIP* , *sourcepath* , [*maxrecordsize*] , [*srcCSVseparator*] , [*srcCSVterminator*] , [*srcCSVquote*] , *destinationgroup* , *destinationlogicalname* , [*timeout*] , [*espserverIPport*] , [*maxConnections*] , [*allowoverwrite*] , [*replicate*] , [*compress*] , [*sourceCsvEscape*]);

<i>sourceIP</i>	A null-terminated string containing the IP address of the file.
<i>sourcepath</i>	A null-terminated string containing the path and name of the file.
<i>maxrecordsize</i>	Optional. An integer containing the maximum size of the records in the file. If omitted, the default is 4096.
<i>srcCSVseparator</i>	Optional. A null-terminated string containing the CSV field separator. If omitted, the default is '\\,'
<i>srcCSVterminator</i>	Optional. A null-terminated string containing the CSV record separator. If omitted, the default is '\\n,\\r\\n'
<i>srcCSVquote</i>	Optional. A null-terminated string containing the CSV quoted field delimiter. If omitted, the default is '\"'
<i>destinationgroup</i>	A null-terminated string containing the name of the specific supercomputer within the target cluster.
<i>destinationlogical-name</i>	A null-terminated string containing the logical name of the file.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value in the lib_system.ws_fs_server attribute.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is one (1).
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>replicate</i>	Optional. A boolean TRUE or FALSE flag indicating whether to replicate the new file. If omitted, the default is FALSE.
<i>compress</i>	Optional. A boolean TRUE or FALSE flag indicating whether to compress the new file. If omitted, the default is FALSE.
<i>sourceCsvEscape</i>	Optional. A null-terminated string containing the CSV escape characters. If omitted, the default is none.
<i>failIfNoSourceFile</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the spray to fail if no source file is found. If omitted, default is FALSE.
<i>recordStructurePresent</i>	Optional. A boolean TRUE or FALSE flag indicating whether to derive the record structure from the header of the file. If omitted, the default is FALSE.
<i>quotedTerminator</i>	Optional. A boolean TRUE or FALSE flag indicating whether the terminator character can be included in a quoted field. Defaults to TRUE. If FALSE, it allows quicker partitioning of the file (avoiding a complete file scan).

Standard Library Reference
File Movement

<i>encoding</i>	A null-terminated string containing the encoding. Can be set to one of the following: ascii, utf8, utf8n, utf16, utf16le, utf16be, utf32, utf32le, utf32be. If omitted, the default is ascii.
<i>dfuwuid</i>	The definition name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	fSprayVariable returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **SprayVariable** function takes a variable length file from the landing zone and distributes it across the nodes of the destination supercomputer.

Example:

```
STD.File.SprayVariable('10.150.50.14',  
    'c:\\InputData\\MyFile.txt',  
    ''',  
    '400way',  
    'IN:MyFile',  
    -1,  
    'http://10.150.50.12:8010/FileSpray');
```

SprayXML

STD.File.SprayXML(*sourceIP* , *sourcepath* , [*maxrecordsize*] , *srcRowTag* , [*srcEncoding*] , *destinationgroup*, *destinationlogicalname* [*timeout*] [*espserverIPport*] [*maxConnections*] [*allowoverwrite*] [*replicate*] [*compress*])

dfuwuid := **STD.File.fSprayXML**(*sourceIP* , *sourcepath*, [*maxrecordsize*] , *srcRowTag* , [*srcEncoding*] , *destinationgroup*, *destinationlogicalname* , [*timeout*], [*espserverIPport*] , [*maxConnections*], [*allowoverwrite*], [*replicate*], [*compress*]);

<i>sourceIP</i>	A null-terminated string containing the IP address of the file.
<i>sourcepath</i>	A null-terminated string containing the path and name of the file.
<i>maxrecordsize</i>	Optional. An integer containing the maximum size of the records in the file. If omitted, the default is 8192.
<i>srcRowTag</i>	A null-terminated string containing the row delimiting XML tag. Required.
<i>srcEncoding</i>	Optional. A null-terminated string containing the encoding. If omitted, the default is 'utf8'
<i>destinationgroup</i>	A null-terminated string containing the name of the specific supercomputer within the target cluster.
<i>destinationlogical-name</i>	A null-terminated string containing the logical name of the file.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>maxConnections</i>	Optional. An integer specifying the maximum number of connections. If omitted, the default is one (1).
<i>allowoverwrite</i>	Optional. A boolean TRUE or FALSE flag indicating whether to allow the new file to overwrite an existing file of the same name. If omitted, the default is FALSE.
<i>replicate</i>	Optional. A boolean TRUE or FALSE flag indicating whether to replicate the new file. If omitted, the default is FALSE.
<i>compress</i>	Optional. A boolean TRUE or FALSE flag indicating whether to compress the new file. If omitted, the default is FALSE.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the job.
Return:	fSprayXML returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **SprayXML** function takes a well-formed XML file from the landing zone and distributes it across the nodes of the destination supercomputer, producing a well-formed XML file on each node.

Example:

```
STD.File.SprayXML('10.150.50.14','c:\\InputData\\MyFile.txt',,
  'Row',, '400way', 'IN::MyFile', -1,
  'http://10.150.50.12:8010/FileSpray');
```


WaitDfuWorkunit

STD.File.WaitDfuWorkunit(*dfuwuid* [,*timeout*] [,*espserverIPport*])

<i>dfuwuid</i>	A null-terminated string containing the DFU workunit ID (DFUWUID) for the job to wait for. This value is returned by the “leading-f” versions of the Copy, DKC, SprayFixed, SprayVariable, SprayXML, and Despray FileServices functions.
<i>timeout</i>	Optional. An integer value indicating the timeout setting. If omitted, the default is -1. If set to zero (0), execution control returns immediately to the ECL workunit without waiting for the DFU workunit to complete.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/ FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
Return:	WaitDfuWorkunit returns a null-terminated string containing the final status string of the DFU workunit (such as: scheduled, queued, started, aborted, failed, finished, or monitoring).

The **WaitDfuWorkunit** function waits for the specified DFU workunit to finish. Typically that workunit will have been launched with its *timeout* parameter set to zero (0).

Example:

```
STD.File.WaitDfuWorkunit('D20051108-143758');
```

String Handling

CleanAccents

STD.Uni.CleanAccents(*source*)

<i>source</i>	A string containing the data to clean.
Return:	CleanAccents returns a UNICODE value.

The **CleanAccents** function returns the *source* string with all accented characters replaced with unaccented.

Example:

```
UNICODE A := STD.Uni.CleanAccents(u'caf\u00E9'); //café - U+00E9 is lowercase e with acute
//a contains 'cafe'
```

CleanSpaces

STD.Str.CleanSpaces(*source*)

STD.Uni.CleanSpaces(*source*)

<i>source</i>	A string containing the data to clean.
Return:	CleanSpaces returns either a STRING or UNICODE value, as appropriate.

All variations of the **CleanSpaces** function return the *source* string with all instances of multiple adjacent space characters (2 or more spaces together, or a tab character) reduced to a single space character. It also trims off all leading and trailing spaces.

Example:

```
A := STD.Str.CleanSpaces('ABCDE  ABCDE');  
  //A contains 'ABCDE ABCDE'  
UNICODE C := STD.Uni.CleanSpaces(U'ABCDE ABCDE');  //C contains U'ABCDE ABCDE'
```

CompareAtStrength

STD.Uni.CompareAtStrength(*source1*, *source2*, *strength*)

STD.Uni.LocaleCompareAtStrength(*source1*,*source2*,*locale*,*strength*)

<i>source1</i>	A string containing the data to compare.
<i>source2</i>	A string containing the data to compare.
<i>strength</i>	An integer value indicating how to compare. Valid values are:
	1 ignores accents and case, differentiating only between letters.
	2 ignores case but differentiates between accents.
	3 differentiates between accents and case but ignores e.g. differences between Hiragana and Katakana
	4 differentiates between accents and case and e.g. Hiragana/Katakana, but ignores e.g. Hebrew cantillation marks
	5 differentiates between all strings whose canonically decomposed forms (NFD—Normalization Form D) are non-identical
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	CompareAtStrength returns an INTEGER value.

The **CompareAtStrength** functions return zero (0) if the *source1* and *source2* strings contain the same data, ignoring any differences in the case of the letters. These functions return negative one (-1) if *source1* < *source2* or positive one (1) if *source1* > *source2*.

Example:

```
base := u'caf\u00E9'; // U+00E9 is lowercase e with acute
prim := u'coffee shop'; // 1st difference, different letters
seco := u'cafe'; // 2nd difference, accents (no acute)
tert := u'Caf\u00C9'; // 3rd, caps (U+00C9 is u/c E + acute)

A := STD.Uni.CompareAtStrength(base, prim, 1) != 0;
// base and prim differ at all strengths

A := STD.Uni.CompareAtStrength(base, seco, 1) = 0;
// base and seco same at strength 1 (differ only at strength 2)

A := STD.Uni.CompareAtStrength(base, tert, 1) = 0;
// base and tert same at strength 1 (differ only at strength 3)

A := STD.Uni.CompareAtStrength(base, seco, 2) != 0;
// base and seco differ at strength 2

A := STD.Uni.CompareAtStrength(base, tert, 2) = 0;
// base and tert same at strength 2 (differ only at strength 3)

A := STD.Uni.CompareAtStrength(base, seco, 3) != 0;
// base and seco differ at strength 2

A := STD.Uni.CompareAtStrength(base, tert, 3) != 0;
// base and tert differ at strength 3
```

CompareIgnoreCase

STD.Str.CompareIgnoreCase(*source1*,*source2*)

STD.Uni.CompareIgnoreCase(*source1*,*source2*)

STD.Uni.LocaleCompareIgnoreCase(*source1*,*source2*, *locale*)

<i>source1</i>	A string containing the data to compare.
<i>source2</i>	A string containing the data to compare.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	CompareIgnoreCase returns an INTEGER value.

The **CompareIgnoreCase** functions return zero (0) if the *source1* and *source2* strings contain the same data, ignoring any differences in the case of the letters. These functions return negative one (-1) if *source1* < *source2* or positive one (1) if *source1* > *source2*.

Example:

```
A := STD.Str.CompareIgnoreCase('ABCDE', 'abcde');  
//A contains 0 -- they "match"  
  
B := STD.Str.CompareIgnoreCase('ABCDE', 'edcba');  
//B contains -1 -- they do not "match"
```

Contains

STD.Str.Contains(*source*, *pattern*, *nocase*)

STD.Uni.Contains(*source*, *pattern*, *nocase*)

<i>source</i>	A string containing the data to search.
<i>pattern</i>	A string containing the characters to compare. An empty string ('') always returns true.
<i>nocase</i>	A boolean true or false indicating whether to ignore the case.
Return:	Contains returns a BOOLEAN value.

The **Contains** functions return true if all the characters in the *pattern* appear in the *source*, otherwise they return false.

Example:

```
A := STD.Str.Contains(
  'the quick brown fox jumps over the lazy dog',
  'ABCdefghijklmnopqrstuvwxyz', true); //returns TRUE

B := STD.Str.Contains(
  'the speedy ochre vixen leapt over the indolent retriever',
  'abcdefghijklmnopqrstuvwxyz', false); //returns FALSE -- 'z' is missing
```

CountWords

STD.Str.CountWords(*source*, *separator*)

<i>source</i>	A string containing the words to count.
<i>separator</i>	A string containing the word delimiter to use.
Return:	CountWords returns an integer value.

The **CountWords** function returns the number of words in the *source* string based on the specified *separator*.

Example:

```
IMPORT Std;

str1 := 'a word a day keeps the doctor away';
str2 := 'a|word|a|day|keeps|the|doctor|away';

output(LENGTH(TRIM(Str1,LEFT,RIGHT)) - LENGTH(TRIM(Str1,ALL)) + 1);
//finds eight words by removing spaces
STD.Str.CountWords(str1,' '); //finds eight words based on space delimiter
STD.Str.CountWords(str2,'|'); //finds eight words based on bar delimiter
```


EditDistance

STD.Str.EditDistance(*string1*, *string2*)

STD.Uni.EditDistance(*string1*, *string2*, *locale*)

<i>string1</i>	The first of a pair of strings to compare.
<i>string2</i>	The second of a pair of strings to compare.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	EditDistance returns an UNSIGNED4 value.

The **EditDistance** function returns a standard Levenshtein distance algorithm score for the edit distance between *string1* and *string2*. This score reflects the minimum number of operations needed to transform *string1* into *string2*.

Example:

```
STD.Str.EditDistance('CAT','CAT'); //returns 0
STD.Str.EditDistance('CAT','BAT'); //returns 1
STD.Str.EditDistance('BAT','BAIT'); //returns 1
STD.Str.EditDistance('CAT','BAIT'); //returns 2
```

EditDistanceWithinRadius

STD.Str.EditDistanceWithinRadius(*string1*, *string2*, *radius*)

STD.Uni.EditDistanceWithinRadius(*string1*, *string2*, *radius*, *locale*)

<i>string1</i>	The first of a pair of strings to compare.
<i>string2</i>	The second of a pair of strings to compare.
<i>radius</i>	An integer specifying the maximum acceptable edit distance.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	EditDistanceWithinRadius returns a BOOLEAN value.

The **EditDistanceWithinRadius** function returns TRUE if the edit distance between *string1* and *string2* is within the *radius*. The two strings are trimmed before comparison.

Example:

```
IMPORT STD;
STD.Str.EditDistance('CAT','BAIT');           //returns 2

STD.Str.EditDistanceWithinRadius('CAT','BAIT',1); //returns FALSE
STD.Str.EditDistanceWithinRadius('CAT','BAIT',2); //returns TRUE
```

EndsWith

STD.Str.EndsWith(*source*, *suffix*)

<i>source</i>	The string to search.
<i>suffix</i>	The string to find.
Return:	EndsWith returns a BOOLEAN value.

The **EndsWith** function returns TRUE if the *source* ends with the text in the *suffix* parameter.

Example:

```
IMPORT STD;  
STD.Str.EndsWith('a word away','away'); //returns TRUE  
STD.Str.EndsWith('a word a way','away'); //returns FALSE
```

EqualIgnoreCase

STD.Str.EqualIgnoreCase(*source1*,*source2*)

<i>source1</i>	A string containing the data to compare.
<i>source2</i>	A string containing the data to compare.
Return:	EqualIgnoreCase returns a BOOLEAN value.

The **EqualIgnoreCase** function return TRUE if the *source1* and *source2* strings contain the same data, ignoring any differences in the case of the letters.

Example:

```
A := STD.Str.EqualIgnoreCase('ABCDE','abcde');  
//A contains TRUE -- they "match"  
  
B := STD.Str.CompareIgnoreCase('ABCDE','edcba');  
//B contains FALSE -- they do not "match"
```

ExcludeFirstWord

STD.Str.ExcludeFirstWord(*text*)

<i>text</i>	A string containing words separated by whitespace.
Return:	ExcludeFirstWord returns a STRING value.

The **ExcludeFirstWord** function returns the *text* string with the first word removed. Words are separated by one or more whitespace characters. Whitespace before the first word is also removed.

Example:

```
A := STD.Str.ExcludeFirstWord('The quick brown fox');  
//A contains 'quick brown fox'
```

ExcludeLastWord

STD.Str.ExcludeLastWord(*text*)

<i>text</i>	A string containing words separated by whitespace.
Return:	ExcludeLastWord returns a STRING value.

The **ExcludeLastWord** function returns the *text* string with the last word removed. Words are separated by one or more whitespace characters. Whitespace after the last word is also removed.

Example:

```
A := STD.Str.ExcludeLastWord('The quick brown fox');  
//A contains 'The quick brown'
```

ExcludeNthWord

STD.Str.ExcludeNthWord(*text*, *n*)

<i>text</i>	A string containing words separated by whitespace.
<i>n</i>	A integer containing the ordinal position of the word to remove.
Return:	ExcludeNthWord returns a STRING value.

The **ExcludeNthWord** function returns the *text* string with the *n*th word removed. Words are separated by one or more whitespace characters. Whitespace after the *n*th word is also removed (along with whitespace before, if *n*=1).

Example:

```
A := STD.Str.ExcludeNthWord('The quick brown fox',2);  
//A contains 'The brown fox'
```

Extract

STD.Str.Extract(*source*, *instance*)

STD.Uni.Extract(*source*, *instance*)

<i>source</i>	A string containing a comma-delimited list of data.
<i>instance</i>	An integer specifying the ordinal position of the data item within the <i>source</i> to return.
Return:	Extract returns either a STRING or UNICODE value, as appropriate.

The **Extract** function returns the data at the ordinal position specified by the *instance* within the comma-delimited *source* string.

Example:

```
//all these examples result in 'Success'

A := IF(STD.Str.Extract('AB,CD,,G,E',0) = '',
    'Success',
    'Failure -1');

B := IF(STD.Str.Extract('AB,CD,,G,E',1) = 'AB',
    'Success',
    'Failure -2');

C := IF(STD.Str.Extract('AB,CD,,G,E',2) = 'CD',
    'Success',
    'Failure -3');

D := IF(STD.Str.Extract('AB,CD,,G,E',3) = '',
    'Success',
    'Failure -4');

E := IF(STD.Str.Extract('AB,CD,,G,E',4) = 'G',
    'Success',
    'Failure -5');

F := IF(STD.Str.Extract('AB,CD,,G,E',5) = 'E',
    'Success',
    'Failure -6');

G := IF(STD.Str.Extract('AB,CD,,G,E',6) = '',
    'Success',
    'Failure -7');
```


Filter

STD.Str.Filter(*source*, *filterstring*)

STD.Uni.Filter(*source*, *filterstring*)

<i>source</i>	A string containing the data to filter.
<i>filterstring</i>	A string containing the characters to use as the filter.
Return:	Filter returns a STRING or UNICODE value, as appropriate.

The **StringFilter** functions return the *source* string with all the characters except those in the *filterstring* removed.

Example:

```
//all these examples result in 'Success'

A := IF(STD.Str.Filter('ADCBE', 'BD') = 'DB',
    'Success',
    'Failure - 1');

B := IF(STD.Str.Filter('ADCBEREBD', 'BDG') = 'DBBD',
    'Success',
    'Failure - 2');

C := IF(STD.Str.Filter('ADCBE', '') = '',
    'Success',
    'Failure - 3');

D := IF(STD.Str.Filter('', 'BD') = '',
    'Success',
    'Failure - 4');

E := IF(STD.Str.Filter('ABCDE', 'EDCBA') = 'ABCDE',
    'Success',
    'Failure - 5');
```

FilterOut

STD.Str.FilterOut(*source*, *filterstring*)

STD.Uni.FilterOut(*source*, *filterstring*)

<i>source</i>	A string containing the data to filter.
<i>filterstring</i>	A string containing the characters to use as the filter.
Return:	FilterOut returns a STRING or UNICODE value, as appropriate.

The **FilterOut** functions return the *source* string with all the characters in the *filterstring* removed.

Example:

```
//all these examples result in 'Success'

A := IF(STD.Str.FilterOut('ABCDE', 'BD') = 'ACE',
    'Success',
    'Failure - 1');

B := IF(STD.Str.FilterOut('ABCDEABCDE', 'BD') = 'ACEACE',
    'Success',
    'Failure - 2');

C := IF(STD.Str.FilterOut('ABCDEABCDE', '') = 'ABCDEABCDE',
    'Success',
    'Failure - 3');

D := IF(STD.Str.FilterOut('', 'BD') = '',
    'Success',
    'Failure - 4');
```

Find

STD.Str.Find(*source*, *target*, *instance*)

STD.Uni.Find(*source*, *target*, *instance*)

STD.Uni.LocaleFind(*source*, *target*, *instance*, *locale*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the substring to search for.
<i>instance</i>	An integer specifying which occurrence of the <i>target</i> to find.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	Find returns an INTEGER value.

The **Find** functions return the beginning index position within the *source* string of the specified *instance* of the *target* string. If the *target* is not found or the specified *instance* is greater than the number of occurrences of the *target* in the *source*, **Find** returns zero (0).

Example:

```
A := IF(STD.Str.Find('ABCDE', 'BC', 1) = 2,  
    'Success',  
    'Failure - 1'); //success  
  
B := IF(STD.Str.Find('ABCDEABCDE', 'BC', 2) = 7,  
    'Success',  
    'Failure - 2'); //success  
  
C := IF(STD.Str.Find('ABCDEABCDE', '') = 0,  
    'Success',  
    'Failure - 3'); //syntax error, missing 3rd parameter  
  
D := IF(STD.Str.Find('', 'BD', 1) = 0,  
    'Success',  
    'Failure - 4'); //success
```

FindCount

STD.Str.FindCount(*source*, *target*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the substring to search for.
Return:	StringFindCount returns an INTEGER value.

The **FindCount** function returns the number of non-overlapping instances of the *target* string within the *source* string.

Example:

```
A := IF(STD.Str.FindCount('ABCDE', 'BC') = 1,  
    'Success',  
    'Failure - 1'); //success  
  
B := IF(STD.Str.FindCount('ABCDEABCDE', 'BC') = 1,  
    'Success',  
    'Failure - 1'); //failure
```

FindAtStrength

STD.Uni.LocaleFindAtStrength(*source,target,instance,locale,strength*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the substring to search for.
<i>instance</i>	An integer specifying which occurrence of the <i>target</i> to find.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
<i>strength</i>	An integer value indicating how to compare. Valid values are: 1 ignores accents and case, differentiating only between letters 2 ignores case but differentiates between accents. 3 differentiates between accents and case but ignores e.g. differences between Hiragana and Katakana 4 differentiates between accents and case and e.g. Hiragana/Katakana, but ignores e.g. Hebrew cantellation marks 5 differentiates between all strings whose canonically decomposed forms (NFD—Normalization Form D) are non-identical
Return:	FindAtStrength returns an INTEGER value.

The **FindAtStrength** function returns the beginning index position within the *source* string of the specified *instance* of the *target* string. If the *target* is not found or the specified *instance* is greater than the number of occurrences of the *target* in the *source*, StringFind returns zero (0).

Example:

```
base := u'caf\u00E9'; // U+00E9 is lowercase e with acute
prim := u'coffee shop'; // 1st difference, different letters
seco := u'cafe'; // 2nd difference, accents (no acute)
tert := u'Caf\u00C9'; // 3rd, caps (U+00C9 is u/c E + acute)
search := seco + tert + base;
STD.Uni.LocaleFindAtStrength(search, base, 1, 'fr', 1) = 1;
// at strength 1, base matches seco (only secondary diffs)
STD.Uni.LocaleFindAtStrength(search, base, 1, 'fr', 2) = 5;
// at strength 2, base matches tert (only tertiary diffs)
STD.Uni.LocaleFindAtStrength(search, base, 1, 'fr', 3) = 9;
// at strength 3, base doesn't match either seco or tert
STD.Uni.LocaleFindAtStrength(u'le caf\u00E9 vert',
    u'cafe', 1, 'fr', 2) = 4;
// however, an accent on the source,
STD.Uni.LocaleFindAtStrength(u'le caf\u00E9 vert',
    u'cafe', 1, 'fr', 3) = 4;
// rather than on the pattern,
STD.Uni.LocaleFindAtStrength(u'le caf\u00E9 vert',
    u'cafe', 1, 'fr', 4) = 4;
// is ignored at strengths up to 4,
STD.Uni.LocaleFindAtStrength(u'le caf\u00E9 vert',
    u'cafe', 1, 'fr', 5) = 0;
// and only counts at strength 5
```

FindAtStrengthReplace

STD.Uni.LocaleFindAtStrengthReplace(*source*, *target*, *replacement*, *locale*, *strength*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the substring to search for.
<i>replacement</i>	A string containing the replacement data.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
<i>strength</i>	An integer value indicating how to compare. Valid values are:
	1 ignores accents and case, differentiating only between letters.
	2 ignores case but differentiates between accents.
	3 differentiates between accents and case but ignores e.g. differences between Hiragana and Katakana
	4 differentiates between accents and case and e.g. Hiragana/Katakana, but ignores e.g. Hebrew cantillation marks
	5 differentiates between all strings whose canonically decomposed forms (NFD—Normalization Form D) are non-identical
Return:	FindAtStrengthReplace returns a UNICODE value.

The **FindAtStrengthReplace** functions return the *source* string with the *replacement* string substituted for all instances of the *target* string. If the *target* string is not in the *source* string, it returns the *source* string unaltered.

Example:

```
STD.Uni.LocaleFindAtStrengthReplace(u'e\u00E8E\u00C9eE',  
    u'e\u00E9', u'xyz', 'fr', 1) = u'xyzxyzxyz';  
STD.Uni.LocaleFindAtStrengthReplace(u'e\u00E8E\u00C9eE',  
    u'e\u00E9', u'xyz', 'fr', 2) = u'e\u00E8xyzeE';  
STD.Uni.LocaleFindAtStrengthReplace(u'e\u00E8E\u00C9eE',  
    u'e\u00E9', u'xyz', 'fr', 3) = u'e\u00E8E\u00C9eE';
```

FindReplace

STD.Str.FindReplace(*source*, *target*, *replacement*)

STD.Uni.FindReplace(*source*, *target*, *replacement*)

STD.Uni.LocaleFindReplace(*source*, *target*, *replacement*, *locale*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the substring to search for.
<i>replacement</i>	A string containing the replacement data.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	FindReplace returns a STRING or UNICODE value, as appropriate.

The **FindReplace** functions return the *source* string with the *replacement* string substituted for all instances of the *target* string . If the *target* string is not in the *source* string, it returns the *source* string unaltered.

Example:

```
A := STD.Str.FindReplace('ABCDEABCDE', 'BC', 'XY');
//A contains 'AXYDEAXYDE'
A := STD.Uni.FindReplace(u'abcde', u'a', u'AAAAA');
//A contains u'AAAAAbcde'
A := STD.Uni.FindReplace(u'aaaaa', u'aa', u'b');
//A contains u'bba'
A := STD.Uni.FindReplace(u'aaaaaa', u'aa', u'b');
//A contains u'bbb'
A := STD.Uni.LocaleFindReplace(u'gh\u0131klm', u'hyk', u'XxXxX', 'lt');
//A contains u'gXxXxXlm'
A := STD.Uni.LocaleFindReplace(u'gh\u0131klm', u'hyk', u'X', 'lt');
//A contains u'gXlm'
```

FromHexPairs

STD.Str.FromHexPairs(*source*)

<i>source</i>	The string containing the hex pairs to process.
Return:	FromHexPairs returns a data value with each byte created from a pair of hex digits.

The **FromHexPairs** function returns a data value with each byte created from a pair of hex digits.

Example:

```
A := STD.Str.FromHexPairs('0001FF80');
```


GetNthWord

STD.Str.GetNthWord(*source*, *instance*)

STD.Uni.GetNthWord (*source*, *instance* [, *locale*])

<i>source</i>	A string containing the space-delimited words.
<i>instance</i>	An integer specifying the word to return.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	GetNthWord returns a string value.

The **GetNthWord** function returns the word in the *instance* position in the *source* string.

Example:

```
IMPORT Std;  
  
str1 := 'a word a day keeps the doctor away';  
  
STD.Str.GetNthWord(str1,2);    //returns "word"
```

RemoveSuffix

STD.Str.RemoveSuffix(*source*, *suffix*)

<i>source</i>	The string to search.
<i>suffix</i>	The ending string to remove.
Return:	RemoveSuffix returns a string value.

The **RemoveSuffix** function returns the *source* string with the ending text in the *suffix* parameter removed. If the *source* string does not end with the *suffix*, then the *source* string is returned unchanged.

Example:

```
IMPORT STD;
STD.Str.RemoveSuffix('a word away', 'away'); //returns 'a word'
STD.Str.RemoveSuffix('a word a way', 'away'); //returns 'a word a way'
```

Repeat

STD.Str.Repeat(*text*, *n*)

<i>text</i>	The string to be repeated (maximum length is 255 characters).
<i>n</i>	The number of repetitions.
Return:	Repeat returns a STRING containing <i>n</i> concatenations of the string <i>text</i> ..

The **Repeat** function returns the *source* string repeated *n* times.

Example:

```
A := STD.Str.Repeat('ABC',3); //A contains 'ABCABCABC'
```

Reverse

STD.Str.Reverse(*source*)

STD.Uni.Reverse(*source*)

<i>source</i>	A string containing the data to reverse.
Return:	Reverse returns a STRING or UNICODE value, as appropriate.

The **Reverse** functions return the *source* string with all characters in reverse order.

Example:

```
A := STD.Str.Reverse('ABCDE'); //A contains 'EDCBA'
```

SplitWords

STD.Str.SplitWords(*source*, *separator* [, *allowblank*])

<i>source</i>	A string containing the words to extract.
<i>separator</i>	A string containing the word delimiter to use.
<i>allowblank</i>	Optional. If TRUE, specifies allowing blank items in the result. If omitted, the default is FALSE.
Return:	SplitWords returns a SET OF STRING values.

The **SplitWords** function returns the list of words in the *source* string split out by the specified *separator*.

Example:

```
IMPORT Std;

str1 := 'a word a day keeps the doctor away';
str2 := 'a|word|a|day|keeps|the|doctor|away';

STD.Str.SplitWords(str1, ' ');
//returns ['a', 'word', 'a', 'day', 'keeps', 'the', 'doctor', 'away']

STD.Str.SplitWords(str2, '|');
//returns ['a', 'word', 'a', 'day', 'keeps', 'the', 'doctor', 'away']
```

SubstituteExcluded

STD.Str.SubstituteExcluded(*source*, *target*, *replacement*)

STD.Uni.SubstituteExcluded(*source*, *target*, *replacement*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the characters to search for.
<i>replacement</i>	A string containing the replacement character as its first character.
Return:	SubstituteExcluded returns a STRING or UNICODE value, as appropriate.

The **SubstituteExcluded** functions return the *source* string with the *replacement* character substituted for all characters except those in the *target* string. If the *target* string is not in the *source* string, it returns the *source* string with all characters replaced by the *replacement* character.

Example:

```
IMPORT STD;
A := STD.Uni.SubstituteExcluded(u'abcdeabcdec', u'cd', u'x');
//A contains u'xxcdxxxcxdc';
```

SubstituteIncluded

STD.Str.SubstituteIncluded(*source*, *target*, *replacement*)

STD.Uni.SubstituteIncluded(*source*, *target*, *replacement*)

<i>source</i>	A string containing the data to search.
<i>target</i>	A string containing the characters to search for.
<i>replacement</i>	A string containing the replacement character as its first character.
Return:	SubstituteIncluded returns a STRING or UNICODE value, as appropriate.

The **SubstituteIncluded** functions return the *source* string with the *replacement* character substituted for all characters that exist in both the *source* and the *target* string. If no *target* string characters are in the *source* string, it returns the *source* string unaltered.

Example:

```
IMPORT STD;
A := STD.Uni.SubstituteIncluded(u'abcde', u'cd', u'x');
//A contains u'abxxe';
B := STD.Str.SubstituteIncluded('abcabc', 'ac', 'yz');
//B contains 'ybyyby'
```

StartsWith

STD.Str.StartsWith(*source*, *prefix* **)**

<i>source</i>	The string to search.
<i>prefix</i>	The string to find.
Return:	StartsWith returns a BOOLEAN value.

The **StartsWith** function returns TRUE if the *source* starts with the text in the *prefix* parameter.

Example:

```
IMPORT STD;
STD.Str.StartsWith('a word away', 'a word');    //returns TRUE
STD.Str.StartsWith('a word away', 'aword');    //returns FALSE
```


ToHexPairs

STD.Str.ToHexPairs(*source*)

<i>source</i>	The data value that should be expanded as a sequence of hex pairs.
Return:	ToHexPairs returns a string containing a sequence of hex pairs.

The **ToHexPairs** function Converts the data value to a sequence of hex pairs.

Example:

```
A := STD.Str.ToHexPairs(D'\000\001\377\200');
```

ToLowerCase

STD.Str.ToLowerCase(*source*)

STD.Uni.ToLowerCase(*source*)

STD.Uni.LocaleToLowerCase(*source*, *locale*)

<i>source</i>	A string containing the data to change case.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	ToLowerCase returns a STRING or UNICODE value, as appropriate.

The **ToLowerCase** functions return the *source* string with all upper case characters converted to lower case.

Example:

```
A := STD.Str.ToLowerCase('ABCDE'); //A contains 'abcde'
```

ToTitleCase

STD.Str.ToTitleCase(*source*)

STD.Uni.ToTitleCase(*source*)

STD.Uni.LocaleToTitleCase(*source*, *locale*)

<i>source</i>	A string containing the data to change case.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	ToTitleCase returns a STRING or UNICODE value, as appropriate.

The **ToTitleCase** functions return the *source* string with the first letter of each word in upper case and all other letters lower cased.

Example:

```
A := STD.Str.ToTitleCase('ABCDE ABCDE '); //A contains 'Abcde Abcde'
B := STD.Str.ToTitleCase('john smith-jones'); //B contains 'John Smith-Jones'
```

ToUpperCase

STD.Str.ToUpperCase(*source*)

STD.Uni.ToUpperCase(*source*)

STD.Uni.LocaleToUpperCase(*source*, *locale*)

<i>source</i>	A string containing the data to change case.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	ToUpperCase returns a STRING value.

The **ToUpperCase** functions return the *source* string with all lower case characters converted to upper case.

Example:

```
A := STD.Str.ToUpperCase('abcde');  
//A contains 'ABCDE'
```

Translate

STD.Str.Translate(*source*, *search*, *replacement*)

<i>source</i>	A string containing the characters to search.
<i>search</i>	A string containing the characters to be replaced by characters in the <i>replacement</i> string.
<i>replacement</i>	A string containing the characters to act as replacements.
Return:	Translate returns a STRING value.

The **Translate** functions return the *source* string with the *replacement* character substituted for all characters in the *source* string. The *search* string characters are replaced by the characters in the equivalent position in the *replacement* string.

If no *search* string characters are in the *source* string, it returns the *source* string unaltered.

Example:

```
IMPORT STD;  
A := STD.Str.Translate('abcabc','ca','yz'); //A contains 'zbyzby'
```

WildMatch

STD.Str.WildMatch(*source*, *pattern*, *nocase*)

STD.Uni.WildMatch(*source*, *pattern*, *nocase*)

<i>source</i>	A string containing the data to search.
<i>pattern</i>	A string containing the wildcard expression to match. Valid wildcards are ? (single character) and * (multiple character).
<i>nocase</i>	A boolean true or false indicating whether to ignore the case.
Return:	WildMatch returns a BOOLEAN value.

The **WildMatch** function returns TRUE if the *pattern* matches the *source*.

The case-insensitive version of the Unicode WildMatch has been optimized for speed over accuracy. For accurate case-folding, you should either use the Unicode ToUpperCase function explicitly and then a case-sensitive the Unicode WildMatch, or use REGEXFIND.

Example:

```
STD.Str.wildmatch('abcdeabcdec', 'a?c*', false) = TRUE;
```

WordCount

STD.Str.WordCount(*source*)

STD.Uni.WordCount(*source* [, *locale*])

<i>source</i>	A string containing the words to count. Words are delimited by spaces.
<i>locale</i>	A null-terminated string containing the language and country code to use to determine correct sort order and other operations.
Return:	WordCount returns an integer value.

The **WordCount** function returns the number of words in the *source* string.

Example:

```
IMPORT Std;

str1 := 'a word a day keeps the doctor away';

output(LENGTH(TRIM(Str1,LEFT,RIGHT)) - LENGTH(TRIM(Str1,ALL)) + 1);
//finds eight words by removing spaces

STD.Str.WordCount(str1); //finds eight words based on space delimiter
```

Metaphone Support

These functions provide a means to implement Double Metaphone or Metaphone 3 phonetic encoding or fuzzy-match algorithms which return a primary code, a secondary code, or both for a given string.

Primary

STD.Metaphone.Primary(source)

STD.Metaphone3.Primary(source)

<i>source</i>	The string to process.
Return:	Primary returns a STRING value.

The **Primary** function returns a textual representation of the source data, similar to a Soundex code. This function returns the first return value from the Double Metaphone algorithm.

The **Metaphone3.Primary** function uses the newer Metaphone 3 libraries which improve phonetic encoding of English words, non-English words familiar to Americans, and first and last names commonly found in the United States (Enterprise Edition only).

Example:

```
r := RECORD
  STRING source;
  STRING M1;
  STRING M2;
  STRING Mboth;
END;

r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone.Primary( L.LastName );
  SELF.M2     := STD.Metaphone.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone.Double( L.LastName );
END;

// Example using Metaphone 3 (available in Enterprise Edition)
/*
r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone3.Primary( L.LastName );
  SELF.M2     := STD.Metaphone3.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone3.Double( L.LastName );
END;
*/

ds := PROJECT(ProgGuide.Person.File,XF(LEFT));

COUNT(ds);
COUNT(ds(M1 <> M2));
OUTPUT(ds);
OUTPUT(ds(M1 <> M2));
```

Secondary

STD.Metaphone.Secondary(source)

STD.Metaphone3.Secondary(source)

<i>source</i>	The string to process.
Return:	Secondary returns a STRING value.

The **Secondary** function returns a textual representation of the source data, similar to a Soundex code. This function returns the second return value from the Double Metaphone algorithm.

The **Metaphone3.SecondaryPrimary** function uses the newer Metaphone 3 libraries which improve phonetic encoding of English words, non-English words familiar to Americans, and first and last names commonly found in the United States (Enterprise Edition only).

Example:

```
r := RECORD
  STRING source;
  STRING M1;
  STRING M2;
  STRING Mboth;
END;

r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone.Primary( L.LastName );
  SELF.M2     := STD.Metaphone.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone.Double( L.LastName );
END;

// Example using Metaphone 3 (available in Enterprise Edition)
/*
r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone3.Primary( L.LastName );
  SELF.M2     := STD.Metaphone3.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone3.Double( L.LastName );
END;
*/

ds := PROJECT(ProgGuide.Person.File,XF(LEFT));

COUNT(ds);
COUNT(ds(M1 <> M2));
OUTPUT(ds);
OUTPUT(ds(M1 <> M2));
```

Double

STD.Metaphone.Double(source)

STD.Metaphone3.Double(source)

<i>source</i>	The string to process.
Return:	Double returns a STRING value.

The **Double** function returns a textual representation of the *source* data, similar to a Soundex code. This function returns both return values from the Double Metaphone algorithm, concatenating the two into a single result string.

The **Metaphone3.Double** function uses the newer Metaphone 3 libraries which improve phonetic encoding of English words, non-English words familiar to Americans, and first and last names commonly found in the United States (Enterprise Edition only).

Example:

```
r := RECORD
  STRING source;
  STRING M1;
  STRING M2;
  STRING Mboth;
END;

r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone.Primary( L.LastName );
  SELF.M2     := STD.Metaphone.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone.Double( L.LastName );
END;

// Example using Metaphone 3 (available in Enterprise Edition)
/*
r XF(ProgGuide.Person.File L) := TRANSFORM
  SELF.source := L.LastName;
  SELF.M1     := STD.Metaphone3.Primary( L.LastName );
  SELF.M2     := STD.Metaphone3.Secondary( L.LastName );
  SELF.Mboth  := STD.Metaphone3.Double( L.LastName );
END;
*/

ds := PROJECT(ProgGuide.Person.File,XF(LEFT));

COUNT(ds);
COUNT(ds(M1 <> M2));
OUTPUT(ds);
OUTPUT(ds(M1 <> M2));
```

Date and Time Handling

Date Data Types

STD.Date.Date_rec

STD.Date.Date_t

STD.Date.Days_t

Date_rec	A RECORD structure containing three fields, and INTEGER2 year, an UNSIGNED1 month, and an UNSIGNED1 day.
Date_t	An UNSIGNED4 containing a date value in YYYYMMDD format.
Days_t	An UNSIGNED4 containing a date value representing the number of elapsed days since a particular base date. This number can be the number of days in the common era (January 1, 1AD = 1) based on either the Julian or Gregorian calendars, or the number of elapsed days since the Gregorian calendar's January 1, 1900 (January 1, 1900 = 1).

The three Date data types defined in the Date Standard Library are:

```
// A record structure with the different elements separated out.
EXPORT Date_rec := RECORD
  INTEGER2   year;
  UNSIGNED1  month;
  UNSIGNED1  day;
END;

//An unsigned number holding a date in the decimal form YYYYMMDD.
//This type does not support dates prior to 1AD
EXPORT Date_t := UNSIGNED4;

//A number of elapsed days. Value depends on the function called.
EXPORT Days_t := UNSIGNED4;
```

See Also: Time Data Types

Time Data Types

STD.Date.Time_rec

STD.Date.Time_t

STD.DateTime_rec

STD.Timestamp_t

Time_rec	A RECORD structure containing three fields, and INTEGER1 hour, an UNSIGNED1 minute, and an UNSIGNED1 second.
Time_t	An UNSIGNED3 holding a time of day in the decimal form HHMMDD.
Seconds_t	An INTEGER8 holding holding a number of seconds. Can be used to represent either a duration or the number of seconds since epoch (Jan 1, 1970).
DateTime_rec	A RECORD structure containing both a Date_rec and a Time_rec
Timestamp_t	An INTEGER8 holding a number of microseconds. Can be used to represent // either a duration or the number of microseconds since epoch (Jan 1, 1970).

The Time data types defined in the Date Standard Library are:

```
// A record structure with the different time elements separated out.
EXPORT Time_rec := RECORD
    UNSIGNED1    hour;
    UNSIGNED1    minute;
    UNSIGNED1    second;
END;

// An unsigned number holding a time of day in the decimal form HHMMDD.
EXPORT Time_t := UNSIGNED3;
// A signed number holding a number of seconds. Can be used to represent either
// a duration or the number of seconds since epoch (Jan 1, 1970).
EXPORT Seconds_t := INTEGER8;

// A record structure with the different date and time elements separated out.
EXPORT DateTime_rec := RECORD
    Date_rec;
    Time_Rec;
END;

// A signed number holding a number of microseconds. Can be used to represent
// either a duration or the number of microseconds since epoch (Jan 1, 1970).
EXPORT Timestamp_t := INTEGER8;
```

See Also: Date Data Types

Year

STD.Date.Year(*date*)

<i>date</i>	A date value in the Date_t format.
Return:	Year returns an INTEGER value.

The **Year** function returns the Year number from the *date* value.

Example:

```
IMPORT STD;
UNSIGNED4 MyDate := 20120101;    //January 1, 2012

Y := STD.Date.Year(MyDate);
//Y contains 2012
```

Month

STD.Date.Month(*date*)

<i>date</i>	A date value in the Date_t format.
Return:	Month returns an INTEGER value in the range of 1 through 12.

The **Month** function returns the month number from the *date* value.

Example:

```
IMPORT STD;
UNSIGNED4 MyDate := 20120101;    //January 1, 2012

M := STD.Date.Month(MyDate);
    //M contains 1, representing January
```


Day

STD.Date.Day(*date*)

<i>date</i>	A date value in the Date_t format.
Return:	Day returns an INTEGER value in the range of 1 through 31.

The **Day** function returns the Day number from the *date* value.

Example:

```
IMPORT STD;
UNSIGNED4 MyDate := 20120101;    //January 1, 2012

D := STD.Date.Day(MyDate);
    //D contains 1, representing the first of the month
```

Hour

STD.Date.Hour(*time*)

<i>time</i>	A time value in the Time_ format.
Return:	Hour returns an INTEGER value representing the hour in the range of 0-23.

The **Hour** function returns the hour from the *time* value.

Example:

```
IMPORT STD;  
MyTime:= STD.Date.CurrentTime(TRUE);    //Local Time  
  
t1 := STD.Date.Hour(MyTime);  
    //t1 contains the hour of the current local time
```

Minute

STD.Date.Minute(*time*)

<i>time</i>	A time value in the Time_ format.
Return:	Minute returns an INTEGER value representing the minute in the range of 0-59.

The **Minute** function returns the minute from the *time* value.

Example:

```
IMPORT STD;
MyTime:= STD.Date.CurrentTime(TRUE);    //Local Time

t1 := STD.Date.Minute(MyTime);
    //t1 contains the minute of the current local time
```

Second

STD.Date.Second(*time*)

<i>time</i>	A time value in the Time_ format.
Return:	Second returns an INTEGER value representing the second in the range of 0-59.

The **Second** function returns the second from the *time* value.

Example:

```
IMPORT STD;
MyTime:= STD.Date.CurrentTime(TRUE);    //Local Time

t1 := STD.Date.Second(MyTime);
    //t1 contains the second of the current local time
```

DateFromParts

STD.Date.DateFromParts(*year*, *month*, *day*)

<i>year</i>	An INTEGER2 year value in the range 0 to 9999.
<i>month</i>	An UNSIGNED1 month value in the range 1 to 12.
<i>day</i>	An UNSIGNED1 day value in the range 1 to 31.
Return:	DateFromParts returns an UNSIGNED4 value.

The **DateFromParts** function returns a Date_t value from the *year*, *month*, and *day* parameters.

Example:

```
IMPORT STD;
INTEGER2 MyYear := 2012;
UNSIGNED1 MyMonth := 1;
UNSIGNED1 MyDay := 1;

D := STD.Date.DateFromParts(MyYear, MyMonth, MyDay);
//D contains 20120101, representing January 1, 2012
```

TimeFromParts

STD.Date.TimeFromParts(*hour*, *minute*, *second*)

<i>hour</i>	An INTEGER1 hour value in the range 0 to 23.
<i>minute</i>	An UNSIGNED1 minute value in the range 0 to 59.
<i>second</i>	An UNSIGNED1 second value in the range 0 to 59.
Return:	TimeFromParts returns a Time_t (An UNSIGNED3 holding a time of day in the decimal form HHMMDD.)

The **TimeFromParts** function returns a Time_t value from the *hour*, *minute*, and *second* parameters.

Example:

```
IMPORT STD;
UNSIGNED1  MyHour   := 23;
UNSIGNED1  MyMinute := 59;
UNSIGNED1  MySecond := 50;

T := STD.Date.TimeFromParts(MyHour, MyMinute, MySecond);
//T contains 235950
```

IsLeapYear

STD.Date.IsLeapYear(*year*)

<i>year</i>	A year value in the INTEGER2 format.
Return:	IsLeapYear returns a BOOLEAN value.

The **IsLeapYear** function returns TRUE if the *year* is a leap year in the Gregorian (or proleptic Gregorian) calendar.

Example:

```
IMPORT STD;
INTEGER2 MyYear := 2012;    //2012

D := STD.Date.IsLeapYear(MyYear);
    //D contains TRUE, 2012 is a leap year
```

IsDateLeapYear

STD.Date.IsDateLeapYear(*date*)

<i>date</i>	A date in Date_t format. (An UNSIGNED4 containing a date value in YYYYMMDD format.)
Return:	IsDateLeapYear returns a BOOLEAN value.

The **IsDateLeapYear** function returns TRUE if the *year* represented in the *date* is a leap year in the Gregorian (or proleptic Gregorian) calendar.

Example:

```
IMPORT STD;
MyDate := 20120112;    //Jan. 12, 2012

D := STD.Date.IsDateLeapYear(MyDate);
    //D contains TRUE, 2012 is a leap year
```


IsValidDate

STD.Date.IsValidDate(*date* , [*yearLowerBound*],[*yearUpperBound*])

<i>date</i>	A date value in the Date_t format.
<i>yearLowerBound</i>	The minimum acceptable year. Optional; defaults to 1800.
<i>yearUpperBound</i>	The maximum acceptable year. Optional; defaults to 2100.
Return:	IsValidDateYear returns a BOOLEAN value.

The **IsValidDate** function returns TRUE if the date is valid, both by range-checking the year and by validating each of the other individual components.

Example:

```
IMPORT STD;
d1 := 19631122;
d2 := 19990230;
firstTest := STD.Date.IsValidDate(d1); //d1 is valid
secondTest := STD.Date.IsValidDate(d2); //d2 is not valid
```

IsValidTime

STD.Date.IsValidTime(*time*)

<i>time</i>	A time value in the Time_t format.
Return:	IsValidTime returns a BOOLEAN value.

The **IsValidDate** function returns TRUE if the time is valid, by validating each of the individual components (hours, minutes, and seconds)..

Example:

```
IMPORT STD;
d1 := 19631122;
d2 := 19990230;
firstTest := STD.Date.IsValidDate(d1); //d1 is valid
secondTest := STD.Date.IsValidDate(d2); //d2 is not valid
```

IsValidGregorianCalendarDate

STD.Date.IsValidGregorianCalendarDate(*date*)

<i>date</i>	A date value in the Date_t format. (An UNSIGNED4 containing a date value in YYYYMMDD format.)
Return:	IsValidGregorianCalendarDateYear returns a BOOLEAN value.

The **IsValidGregorianCalendarDate** function returns TRUE if the date is valid in the Gregorian calendar. The year must be between 1601 and 30827.

Example:

```
IMPORT STD;
d1 := 19991122;
d2 := 15130230;
firstTest := STD.Date.IsValidGregorianCalendarDate(d1); // TRUE
secondTest := STD.Date.IsValidGregorianCalendarDate(d2); // FALSE
```

FromGregorianYMD

STD.Date.FromGregorianYMD(*year*, *month*, *day*)

<i>year</i>	An INTEGER2 year value in the range 0 to 9999.
<i>month</i>	An UNSIGNED1 month value in the range 1 to 12.
<i>day</i>	An UNSIGNED1 day value in the range 1 to 31.
Return:	FromGregorianYMD returns an UNSIGNED4 value.

The **FromGregorianYMD** function returns a Days_t value from the *year*, *month*, and *day* parameters representing the number days since 31st December 1BC in the Gregorian calendar (see The Calendar FAQ by Claus Tondering at <http://www.tondering.dk/claus/calendar.html>).

Example:

```
IMPORT STD;
INTEGER2 MyYear  := 2012;
UNSIGNED1 MyMonth := 1;
UNSIGNED1 MyDay  := 1;

D := STD.Date.FromGregorianYMD(MyYear, MyMonth, MyDay);
//D contains 734503
```

ToGregorianYMD

STD.Date.ToGregorianYMD(*days*)

<i>days</i>	A year value in the Days_t format.
Return:	ToGregorianYMD returns separate values for Year, Month, and Day.

The **ToGregorianYMD** function converts the number days since 31st December 1BC to a date in the Gregorian calendar. It returns a module with three exported values: Year, Month, and Day.

Example:

```
IMPORT STD;
INTEGER2 MyYear := 2012;
UNSIGNED1 MyMonth := 1;
UNSIGNED1 MyDay := 1;

J := STD.Date.FromGregorianYMD(MyYear, MyMonth, MyDay);
//J contains 734503

X := STD.Date.ToGregorianYMD(J);
// X is a module with exported values

Y := X.Year; //Y contains 2012
M := X.Month; //M contains 1
D := X.Day; //D contains 1
```

FromStringToDate

STD.Date.FromStringToDate(*date_text*, *format*)

<i>date_text</i>	The string to be converted
<i>format</i>	The format of the input string. See strftime documentation for details (http://strftime.org/)
return	The date that was matched in the string. Returns 0 if failed to match or if the date components match but the result is an invalid date.

The **FromStringToDate** function converts a string to a Date_t using the relevant string format. The resulting date must be representable within the Gregorian calendar after the year 1600.

Supported characters:

%B	Full month name
%b or %h	Abbreviated month name
%d	Day of month (two digits)
%e	Day of month (two digits, or a space followed by a single digit)
%m	Month (two digits)
%t	Whitespace
%y	year within century (00-99)
%Y	Full year (yyyy)
%j	Julian day (1-366)

Common date formats

American	'%m/%d/%Y'	mm/dd/yyyy	
Euro	'%d/%m/%Y'	dd/mm/yyyy	
Iso format	'%Y-%m-%d'	yyyy-mm-dd	
Iso basic	'%Y%m%d'	yyyymmdd	
	'%d-%b-%Y'	dd-mon-yyyy	e.g., '21-Mar-1954'

Example:

```
IMPORT STD;

D1 := STD.Date.FromStringToDate('19720607', '%Y%m%d');
//D1 contains 19720607
D2 := STD.Date.FromStringToDate('19720007', '%Y%m%d');
//D2 contains 0
D3 := STD.Date.FromStringToDate('4/29/1974', '%m/%d/%Y');
//D3 contains 19740429
D4:= STD.Date.FromStringToDate('29/4/1974', '%d/%m/%Y');
//D4 contains 19740429
```

Today

STD.Date.Today()

Return:	Today returns date_t (an UNSIGNED4 containing a date value in YYYYMMDD format) representing the current date.
---------	---

The **Today** function returns the current date in the local time zone.

Example:

```
IMPORT STD;  
  
D1 := STD.Date.Today();  
    //D1 contains today's date
```

CurrentDate

STD.Date.CurrentDate (*[in_local_time]*)

<i>in_local_time</i>	TRUE if the returned value should be local to the cluster computing the date, FALSE for UTC. Optional, defaults to FALSE.
Return:	Today returns a Date_t representing the current date.

The **CurrentDate** function returns the current date. If the *in_local_time* parameter is TRUE the returned value is local to the cluster computing the date, if FALSE then the UTC value is returned.

Example:

```
IMPORT STD;  
d1 := STD.Date.CurrentDate(True);  
    //d1 contains the current local date
```


CurrentTime

STD.Date.CurrentTime (*[in_local_time]*)

<i>in_local_time</i>	TRUE if the returned value should be local to the cluster computing the time, FALSE for UTC. Optional, defaults to FALSE.
Return:	Today returns a time_t (An UNSIGNED3 holding a time of day in the decimal form HHMMDD.)

The **CurrentTime** function returns the current time. If the *in_local_time* parameter is TRUE the returned value is local to the cluster computing the time, if FALSE then the UTC is returned.

Example:

```
IMPORT STD;
t1 := STD.Date.CurrentTime(True);
//t1 contains the current local time of day
```

DayOfYear

STD.Date.DayOfYear(date)

<i>date</i>	A date value in the Date_t format.
Return:	DayOfYear returns an INTEGER value in the range of 1 through 366.

The **DayOfYear** function returns a number representing the day of the year for the given date. The date must be in the Gregorian calendar after the year 1600.

Example:

```
IMPORT STD;  
D1 := STD.Date.DayOfYear(STD.Date.Today());  
    // D1 contains the day of the year for today's date
```

DaysBetween

STD.Date.DaysBetween(fromDate, toDate)

<i>fromDate</i>	The first date value in Date_t format.
<i>toDate</i>	The last date value in Date_t format.
Return:	DaysBetween returns an INTEGER value of the number of days between the two dates.

The **DayOfYear** function calculate the number of whole months between two dates.

Example:

```
IMPORT STD;
StartDate := 19631122;
numDays := STD.Date.DaysBetween(startDate,STD.Date.Today());
// numDays contains the number of days between the startDate and today's date
```

MonthsBetween

STD.Date.MonthsBetween(fromDate, toDate)

<i>fromDate</i>	The first date value in Date_t format.
<i>toDate</i>	The last date value in Date_t format.
Return:	MonthsBetween returns an INTEGER value of the number of whole months between the two dates.

The **DayOfYear** function calculate the number of whole months between two dates.

Example:

```
IMPORT STD;
StartDate := 19631122;
numMonths := STD.Date.MonthsBetween(startDate,STD.Date.Today());
// numMonths contains the number of months between the startDate and today's date
```

AdjustDate

STD.Date.AdjustDate(*date* , [*year_delta*],[*month_delta*] , [*day_delta*])

<i>date</i>	A date value in the Date_t format.
<i>year_delta</i>	The minimum acceptable year. Optional; defaults to zero.
<i>month_delta</i>	The minimum acceptable year. Optional; defaults to zero.
<i>day_delta</i>	The maximum acceptable year. Optional; defaults to zero.
Return:	AdjustDate returns date_t representing the adjusted date.

The **AdjustDate** function adjusts a date by incrementing or decrementing year, month, and/or day values. The date must be in the Gregorian calendar after the year 1600.

If the new calculated date is invalid then it is normalized according to mktime() rules. For example, 20140130 plus 1 month would be 20140302.

Example:

```
IMPORT std;
inDate :=19631123;
Std.Date.AdjustDate(inDate,5,1,3); //returns 19681226
```

See Also: AdjustCalendar

AdjustCalendar

STD.Date.AdjustCalendar(date , [year_delta],[month_delta] ,[day_delta])

<i>date</i>	A date value in the Date_t format.
<i>year_delta</i>	The minimum acceptable year. Optional; defaults to zero.
<i>month_delta</i>	The minimum acceptable year. Optional; defaults to zero.
<i>day_delta</i>	The maximum acceptable year. Optional; defaults to zero.
Return:	AdjustDate returns date_t representing the adjusted date.

The AdjustCalendar function adjusts a date by incrementing or decrementing months and/or years. The date must be in the Gregorian calendar after the year 1600.

This uses the rule outlined in McGinn v. State, 46 Neb. 427, 65 N.W. 46 (1895):

"The term calendar month, whether employed in statutes or contracts, and not appearing to have been used in a different sense, denotes a period terminating with the day of the succeeding month numerically corresponding to the day of its beginning, less one. If there be no corresponding day of the succeeding month, it terminates with the last day thereof."

Note that day adjustments are performed after year and month adjustments using the preceding rules.

As an example, Jan. 31, 2014 + 1 month results in Feb. 28, 2014; Jan. 31, 2014 + 1 month + 1 day results in Mar. 1, 2014.

Example:

```
IMPORT std;
inDate :=19631123;
Std.Date.AdjustCalendar(inDate,5,1,3); //returns 19681226
```

See Also: AdjustDate

Cluster Handling

Node

STD.System.Thorlib.Node()

Return:	Node returns an UNSIGNED INTEGER4 value.
---------	--

The **Node** function returns the (zero-based) number of the Data Refinery (Thor) or Rapid Data Delivery Engine (Roxie) node.

Example:

```
A := STD.System.ThorLib.Node();
```


Nodes

STD.System.Thorlib.Nodes()

Return:	Nodes returns an UNSIGNED INTEGER4 value.
---------	---

The **Nodes** function returns the number of nodes in the Thor cluster (always returns 1 on hThor and Roxie). This number is the same as the CLUSTERSIZE compile time constant. The Nodes function is evaluated each time it is called, so the choice to use the function versus the constant depends upon the circumstances.

Example:

```
A := STD.System.ThorLib.Nodes();
```

LogicalToPhysical

STD.System.Thorlib.LogicalToPhysical (*filename* [, *createflag*])

<i>filename</i>	A null-terminated string containing the logical name of the file.
<i>createflag</i>	A boolean value indicating whether to create the <i>filename</i> . If omitted, the default is FALSE.
Return:	LogicalToPhysical returns a VARSTRING value.

The **LogicalToPhysical** function (Logical to Physical) returns the physical name of the file represented by the logical *filename*.

Example:

```
A := STD.System.Thorlib.LogicalToPhysical('Fred');
```

DaliServer

STD.System.Thorlib.DaliServer ()

Return:	Dalserver returns a VARSTRING value.
---------	--------------------------------------

The **Dalserver** function returns the IP and port of the system data store (Dali) server for the environment running the workunit.

Example:

```
A := Thorlib.Dalserver();
```

Group

STD.System.Thorlib.Group ()

Return:	Group returns a VARSTRING value.
---------	----------------------------------

The **Group** function returns the name of the node group running the workunit. Not supported on Roxie clusters. This name is used in ECL code to specify the target CLUSTER for an OUTPUT action or a PERSISTed attribute.

Example:

```
A := ThorLib.Group();
```

GetExpandLogicalFileName

ThorLib.GetExpandLogicalFileName (*filename*)

<i>filename</i>	A null-terminated string containing the logical name of the file.
Return:	GetExpandLogicalFileName returns a VARSTRING (null-terminated) value.

The **GetExpandLogicalFileName** function returns a string containing the expanded logical filename (ncluding the default scope, if the filename does not contain a leading tilde), all in lowercase. This is the same value as is used internally by DATASET and OUTPUT.

Example:

```
A := ThorLib.GetExpandLogicalFileName('Fred');
```

Job Handling

WUID

STD.System.Job.WUID ()

Return:	WUID returns a VARSTRING value.
---------	---------------------------------

The **WUID** function returns the workunit identifier of the current job. This is the same as the WORKUNIT compile time constant.

Example:

```
A := STD.System.Job.WUID();
```

Target

STD.System.Job.Target ()

Return:	Target returns a VARSTRING value.
---------	-----------------------------------

The **Target** function returns the name of the cluster running the workunit. Not supported on Roxie clusters. This name is used by #WORKUNIT, the ecl command line utility, or the eclplus command line utility to specify the target cluster for a workunit.

Example:

```
A := STD.System.Job.Target();
```


Name

STD.System.Job.Name ()

Return:	Name returns a VARSTRING value.
---------	---------------------------------

The **Name** function returns the name of the workunit.

Example:

```
A := STD.System.Job.Name();
```

User

STD.System.Job.User ()

Return:	User returns a VARSTRING value.
---------	---------------------------------

The **User** function returns the username of the person running the workunit.

Example:

```
A := STD.System.Job.User();
```

OS

STD.System.Job.OS ()

Return:	OS returns a VARSTRING value.
---------	-------------------------------

The **OS** function returns the operating system (windows or Linux) of the cluster running the workunit.

Example:

```
A := STD.System.Job.OS();
```

Platform

STD.System.Job.Platform ()

Return:	Platform returns a VARSTRING value.
---------	-------------------------------------

The **Platform** function returns the platform name (hthor, thor, or roxie) of the cluster running the workunit.

Example:

```
A := STD.System.Job.Platform();
```

LogString

STD.System.Job.LogString (*message*)

<i>message</i>	A string expression containing the text to place in the log file.
Return:	LogString returns an INTEGER value.

The **LogString** function outputs “USER:” followed by the *message* text to the eclagent or Roxie log file and returns the length of the text written to the file.

Example:

```
A := STD.System.Job.LogString('The text message to log');  
//places USER:The text message to log  
//in the log file
```

File Monitoring

MonitorFile

STD.File.MonitorFile(*event*, [*ip*], *filename*, [*,subdirs*] [*,shotcount*] [*,espserverIPport*])

dfuwuid := **STD.File.fMonitorFile**(*event*, [*ip*], *filename*, [*,subdirs*] [*,shotcount*] [*,espserverIPport*]);

<i>event</i>	A null-terminated string containing the user-defined name of the event to fire when the <i>filename</i> appears. This value is used as the first parameter to the EVENT function.
<i>ip</i>	Optional. A null-terminated string containing the ip address for the file to monitor. This is typically a landing zone. This may be omitted only if the <i>filename</i> parameter contains a complete URL.
<i>filename</i>	A null-terminated string containing the full path to the file to monitor. This may contain wildcard characters (* and ?).
<i>subdirs</i>	Optional. A boolean value indicating whether to include files in sub-directories that match the wildcard mask when the <i>filename</i> contains wildcards. If omitted, the default is false.
<i>shotcount</i>	Optional. An integer value indicating the number of times to generate the event before the monitoring job completes. A negative one (-1) value indicates the monitoring job continues until manually aborted. If omitted, the default is 1.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/FileSpray” appended. If omitted, the default is the value contained in the <code>lib_system.ws_fs_server</code> attribute.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the monitoring job.
Return:	fMonitorFile returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **MonitorFile** function creates a file monitor job in the DFU Server. Once the job is received it goes into a 'monitoring' mode (which can be seen in the ECL Watch DFU Workunit display), which polls at a fixed interval. This interval is specified in the DFU Server's **monitorinterval** configuration setting. The default interval is 900 seconds (15 minutes). If an appropriately named file arrives in this interval it will fire the *event* with the name of the triggering object as the event subtype (see the **EVENT** function).

This process continues until either:

- 1) The *shotcount* number of events have been generated.
- 2) The user aborts the DFU workunit.

The **STD.File.AbortDfuWorkunit** and **STD.File.WaitDfuWorkunit** functions can be used to abort or wait for the DFU job by passing them the returned *dfuwuid*.

Note the following caveats and restrictions:

- 1) Events are only generated when the monitor job starts or subsequently on the polling interval.
- 2) Note that the *event* is generated if the file has been created since the last polling interval. Therefore, the *event* may occur before the file is closed and the data all written. To ensure the file is not subsequently read before it is complete you should use a technique that will preclude this possibility, such as using a separate 'flag' file instead of the file, itself or renaming the file once it has been created and completely written.
- 3) The **EVENT** function's subtype parameter (its 2nd parameter) when monitoring physical files is the full URL of the file, with an absolute IP rather than DNS/netbios name of the file. This parameter cannot be retrieved but can only be used for matching a particular value.

Example:

```
EventName := 'MyFileEvent';  
FileName  := 'c:\\test\\myfile';  
LZ := '10.150.50.14';  
STD.File.MonitorFile(EventName,LZ,FileName);  
OUTPUT('File Found') : WHEN(EVENT(EventName,'*'),COUNT(1));
```


MonitorLogicalFileName

STD.File.MonitorLogicalFileName(*event*, *filename*, [, *shotcount*] [, *espserverIPport*])

dfuwuid := **STD.File.fMonitorLogicalFileName**(*event*, *filename*, [, *shotcount*] [, *espserverIPport*]);

<i>event</i>	A null-terminated string containing the user-defined name of the event to fire when the <i>filename</i> appears. This value is used as the first parameter to the EVENT function.
<i>filename</i>	A null-terminated string containing the name of the logical file in the DFU to monitor.
<i>shotcount</i>	Optional. An integer value indicating the number of times to generate the event before the monitoring job completes. A negative one (-1) value indicates the monitoring job continues until manually aborted. If omitted, the default is 1.
<i>espserverIPport</i>	Optional. A null-terminated string containing the protocol, IP, port, and directory, or the DNS equivalent, of the ESP server program. This is usually the same IP and port as ECL Watch, with “/ FileSpray” appended. If omitted, the default is the value contained in the lib_system.ws_fs_server attribute.
<i>dfuwuid</i>	The attribute name to receive the null-terminated string containing the DFU workunit ID (DFUWUID) generated for the monitoring job.
Return:	fMonitorLogicalFileName returns a null-terminated string containing the DFU workunit ID (DFUWUID).

The **MonitorLogicalFileName** function creates a file monitor job in the DFU Server. Once the job is received it goes into a 'monitoring' mode (which can be seen in the eclwatch DFU Workunit display), which polls at a fixed interval (default 15 mins). If an appropriately named file arrives in this interval it will fire the *event* with the name of the triggering object as the event subtype (see the EVENT function).

This function does not support wildcard characters. To monitor physical files or directories using wildcards, use the MonitorFile function.

This process continues until either:

- 1) The *shotcount* number of events have been generated.
- 2) The user aborts the DFU workunit.

The STD.File.AbortDfuWorkunit and STD.File.WaitDfuWorkunit functions can be used to abort or wait for the DFU job by passing them the returned *dfuwuid*.

Note the following caveats and restrictions:

- 1) If a matching file already exists when the DFU Monitoring job is started, that file will not generate an event. It will only generate an event once the file has been deleted and recreated.
- 2) If a file is created and then deleted (or deleted then re-created) between polling intervals, it will not be seen by the monitor and will not trigger an event.
- 3) Events are only generated on the polling interval.

Example:

```
EventName := 'MyFileEvent';
FileName  := 'test::myfile';
IF (STD.File.FileExists(FileName),
```

```
        STD.File.DeleteLogicalFile(FileName));
STD.File.MonitorLogicalFileName(EventName,FileName);
OUTPUT('File Created') : WHEN(EVENT(EventName,'*'),COUNT(1));

rec := RECORD
    STRING10 key;
    STRING10 val;
END;
afile := DATASET([{'A', '0'}], rec);
OUTPUT(afile,,FileName);
```

Logging

dbglog

STD.System.Log.dbglog (*text*)

<i>text</i>	A string containing the text to write.
Return:	dbglog does not return a value.

The **dbglog** function writes the *text* string to the eclagent.log file for the workunit.

Example:

```
IMPORT STD;  
STD.System.Log.dbglog('Got Here 1');    //write text to log
```

addWorkunitInformation

STD.System.Log.addWorkunitInformation (*text* [, *code*])

<i>text</i>	A string containing the text to write.
<i>code</i>	Optional. The code number to associate with the <i>text</i> . If omitted, the default is zero (0).
Return:	addWorkunitInformation does not return a value.

The **addWorkunitInformation** function writes the *text* string to the eclagent.log file for the workunit, and also displays the *code* and *text* in the Info section of the ECL Watch page for the workunit.

Example:

```
IMPORT STD;
STD.System.Log.addWorkunitInformation('Got Here',1);
//write text to log and display "1: Got Here" as Info
```

addWorkunitWarning

STD.System.Log.addWorkunitWarning (*text* [, *code*])

<i>text</i>	A string containing the text to write.
<i>code</i>	Optional. The code number to associate with the <i>text</i> . If omitted, the default is zero (0).
Return:	addWorkunitWarning does not return a value.

The **addWorkunitWarning** function writes the *text* string to the eclagent.log file for the workunit, and also displays the *code* and *text* in the Syntax Errors toolbox along with the Warnings section of the ECL Watch page for the workunit.

Example:

```
IMPORT STD;  
STD.System.Log.addWorkunitWarning('Got Here',1);  
    //write text to log and display "1: Got Here" in Warnings
```

addWorkunitError

STD.System.Log.addWorkunitError (*text* [, *code*])

<i>text</i>	A string containing the text to write.
<i>code</i>	Optional. The code number to associate with the <i>text</i> . If omitted, the default is zero (0).
Return:	addWorkunitError does not return a value.

The **addWorkunitError** function writes the *text* string to the eclagent.log file for the workunit, and also displays the *code* and *text* in the Syntax Errors toolbox along with the Errors section of the ECL Watch page for the workunit.

Example:

```
IMPORT STD;  
STD.System.Log.addWorkunitError('Got Here',1);  
    //write text to log and display "1: Got Here" in Errors
```

Auditing

Audit

STD.Audit.Audit(*type, message* **)**

<i>type</i>	A string constant containing the type of audit entry. Currently, only INFO is provided.
<i>message</i>	A string containing the audit entry text.
Return:	Audit returns a BOOLEAN value indicating whether it was successful or not.

The **Audit** function writes the *message* into the Windows event log or Linux system log on the ECL Agent computer. The entries can be retrieved from the logs using standard operating system tools.

Example:

```
STD.Audit.Audit('INFO', 'Audit Message');
```

Utilities

GetHostName

result := **STD.System.Util.GetHostName** (*ip*);

<i>ip</i>	A null-terminated string containing the IP address of the remote machine.
Return:	GetHostName returns returns a VARSTRING (null-terminated) value.

The **GetHostName** function does a reverse DNS lookup to return the host name for the machine at the specified *ip* address.

Example:

```
IP    := '10.150.254.6';  
OUTPUT(STD.System.Util.GetHostName(IP));
```

ResolveHostName

result := **STD.System.Util.ResolveHostName** (*host*);

<i>host</i>	A null-terminated string containing the DNS name of the remote machine.
Return:	ResolveHostName returns returns a VARSTRING (null-terminated) value.

The **ResolveHostName** function does a DNS lookup to return the ip address for the specified *host* name.

Example:

```
host := 'dataland_dali.br.seisint.com';  
OUTPUT(STD.System.Util.ResolveHostName(host));
```

CmdProcess

result := **STD.System.Util.CmdProcess** (*program*, *input*);

<i>program</i>	A null-terminated string containing the name of the program to execute. This may include command-line parameters.
<i>input</i>	A string containing the text to pipe into the <i>program</i> through stdin.
Return:	CmdProcess returns returns a STRING value.

The **CmdProcess** function pipes the *input* text to the specified *program*. This is similar to the PIPE built-in function, but limited to simple text input and output.

Example:

```
IMPORT STD;  
  
OUTPUT(STD.System.Util.CmdProcess('cat', 'George Jetson'));
```

GetUniqueInteger

result := **STD.System.Util.GetUniqueInteger** ([*dali*]);

<i>dali</i>	Optional. A null-terminated string containing the ip address of the remote dali to provide the number. If omitted, the default is local.
Return:	GetUniqueInteger returns returns an UNSIGNED8 value.

The **GetUniqueInteger** function returns a number that is unique across all the slave nodes of the specified *dali*.

Example:

```
IMPORT STD;  
  
OUTPUT(STD.System.Util.GetUniqueInteger());
```

Debugging

GetParseTree

STD.System.Debug.GetParseTree ()

Return:	GetParseTree returns a STRING value.
---------	--------------------------------------

The **GetParseTree** function returns a textual representation of the match that occurred, using square brackets (such as: a[b[c]d]) to indicate nesting. This function is only used within the RECORD or TRANSFORM structure that defines the result of a PARSE operation. This function is useful for debugging PARSE operations.

Example:

```
IMPORT STD;

r := {string150 line};
d := dataset([
{'Ge 34:2 And when Shechem the son of Hamor the Hivite, '+
'prince of the country, saw her, he took her, and lay with her, '+
'and defiled her.'},
{'Ge 36:10 These are the names of Esaus sons; Eliphaz the son of '+
'Adah the wife of Esau, Reuel the son of Bashemath the wife of '+
'Esau.'}
],r);
PATTERN ws := [' ', '\t', ',', '*];
PATTERN patStart := FIRST | ws;
PATTERN patEnd := LAST | ws;
PATTERN article := ['A', 'The', 'Thou', 'a', 'the', 'thou'];
TOKEN patWord := PATTERN('[a-zA-Z]+');
TOKEN Name := PATTERN('[A-Z][a-zA-Z]+');
RULE Namet := name OPT(ws 'the' ws name);
PATTERN produced_by := OPT(article ws) ['son of', 'daughter of'];
PATTERN produces_with := OPT(article ws) ['wife of'];
RULE progeny := namet ws ( produced_by | produces_with ) ws namet;
results := RECORD
    STRING LeftName := MATCHTEXT(Namet[1]);
    STRING RightName := MATCHTEXT(Namet[2]);
    STRING LinkPhrase := IF(MATCHTEXT(produced_by[1])<>' ',
        MATCHTEXT(produced_by[1]),
        MATCHTEXT(produces_with[1]));
    STRING Tree := 'Tree: ' + STD.System.Debug.getParseTree();
END;
outfile1 := PARSE(d,line,progeny,results,SCAN ALL);
/* the Tree field output looks like this:
Tree: [namet[name"Shechem"] ws" " produced_by"the son of" ws" " namet[name"Hamor"]]
*/
```


GetXMLParseTree

STD.System.Debug.GetXMLParseTree ()

Return:	GetXMLParseTree returns a STRING value.
---------	---

The **GetXMLParseTree** function returns a textual representation of the match that occurred, using XML tags to indicate nesting. This function is only used within the RECORD or TRANSFORM structure that defines the result of a PARSE operation. This function is useful for debugging PARSE operations.

Example:

```
IMPORT STD;

r := {string150 line};
d := dataset([
{'Ge 34:2 And when Shechem the son of Hamor the Hivite, '+
'prince of the country, saw her, he took her, and lay with her, '+
'and defiled her.'},
{'Ge 36:10 These are the names of Esaus sons; Eliphaz the son of '+
'Adah the wife of Esau, Reuel the son of Bashemath the wife of '+
'Esau.'}
],r);

PATTERN ws := [' ', '\t', ',', '*];
PATTERN patStart := FIRST | ws;
PATTERN patEnd := LAST | ws;
PATTERN article := ['A', 'The', 'Thou', 'a', 'the', 'thou'];
TOKEN patWord := PATTERN('[a-zA-Z]+');
TOKEN Name := PATTERN('[A-Z][a-zA-Z]+');
RULE Namet := name OPT(ws 'the' ws name);
PATTERN produced_by := OPT(article ws) ['son of', 'daughter of'];
PATTERN produces_with := OPT(article ws) ['wife of'];
RULE progeny := namet ws ( produced_by | produces_with ) ws namet;
results := RECORD
    STRING LeftName := MATCHTEXT(Namet[1]);
    STRING RightName := MATCHTEXT(Namet[2]);
    STRING LinkPhrase := IF(MATCHTEXT(produced_by[1])<>' ',
        MATCHTEXT(produced_by[1]),
        MATCHTEXT(produces_with[1]));
    STRING Tree := STD.System.Debug.getXMLParseTree();
END;
outfile1 := PARSE(d,line,progeny,results,SCAN ALL);
/* the Tree field output
looks like this:
<namet>
  <name>Shechem</name>
</namet>
<ws> </ws>
<produced_by>the son of</produced_by>
<ws> </ws>
<namet>
  <name>Hamor</name>
</namet>
*/
```

Sleep

STD.System.Debug.Sleep (*duration*)

<i>duration</i>	An integer value specifying the length of the sleep period, in milliseconds.
Return:	Sleep does not return a value.

The **Sleep** function pauses processing for *duration* milliseconds.

Example:

```
IMPORT STD;  
STD.System.Debug.Sleep(1000);    //pause for one second before continuing
```

msTick

STD.System.Debug.msTick ()

Return:	msTick returns a 4-byte unsigned integer value.
---------	---

The **msTick** function returns elapsed time since its start point, in milliseconds. The start point is undefined, making this function useful only for judging elapsed time between calls to the function by subtracting the latest return value from the earlier. When the return value reaches the maximum value of a 4-byte unsigned integer (2^{32} or 4 Gb), it starts over again at zero (0). This occurs approximately every 49.71 days.

Example:

```
IMPORT STD;
t1 := STD.System.Debug.msTick() : STORED('StartTime'); //get start time

ds1 := DATASET([0,0,0,0,0],
               {UNSIGNED4 RecID,
                UNSIGNED4 Started,
                UNSIGNED4 ThisOne,
                UNSIGNED4 Elapsed,
                UNSIGNED4 RecsProcessed});

RECORDOF(ds1) XF1(ds1 L, integer C) := TRANSFORM
    SELF.RecID := C;
    SELF := L;
END;
ds2 := NORMALIZE(ds1,100000,XF1(LEFT,COUNTER));

RECORDOF(ds1) XF(ds1 L) := TRANSFORM
    SELF.Started := T1;
    SELF.ThisOne := STD.System.Debug.msTick();
    SELF.Elapsed := SELF.ThisOne - SELF.Started;
    SELF := L;
END;

P := PROJECT(ds2,XF(LEFT)) : PERSIST('~RTTEST::TestTick');
R := ROLLUP(P,
            LEFT.Elapsed=RIGHT.Elapsed,
            TRANSFORM(RECORDOF(ds1),
                      SELF.RecsProcessed := RIGHT.RecID - LEFT.RecID,
                      SELF := LEFT));

paws := STD.System.Debug.Sleep(1000); //pause for one second before continuing
SEQUENTIAL(paws,OUTPUT(P, ALL),OUTPUT(R, ALL));
```

Email

SendEmail

STD.System.Email.SendEmail (*sendto*, *subject*, *body*, *server*, *port*, *sender*)

<i>sendto</i>	A null-terminated string containing a comma-delimited list of the addresses of the intended recipients. The validity of the addresses is not checked, so it is the programmer's responsibility to ensure they are all valid.
<i>subject</i>	A null-terminated string containing the subject line.
<i>body</i>	A null-terminated string containing the text of the email to send. This must be character encoding "ISO-8859-1 (latin1)" (the ECL default character set). Text in any other character set must be sent as an attachment (see the STD.System.Email.SendEmailAttachText() function).
<i>server</i>	Optional. A null-terminated string containing the name of the mail server. If omitted, defaults to the value in the SMTPserver environment variable.
<i>port</i>	Optional. An UNSIGNED4 integer value containing the port number. If omitted, defaults to the value in the SMTPport environment variable.
<i>sender</i>	Optional. A null-terminated string containing the address of the sender. If omitted, defaults to the value in the emailSenderAddress environment variable.

The **SendEmail** function sends an email message.

Example:

```
STD.System.Email.SendEmail( 'me@mydomain.com', 'testing 1,2,3', 'this is a test message');
```

SendEmailAttachData

STD.System.Email.SendEmailAttachData (*sendto*, *subject*, *body*, *attachment*, *mimetype*, *filename*, *server*, *port*, *sender*)

<i>sendto</i>	A null-terminated string containing a comma-delimited list of the addresses of the intended recipients. The validity of the addresses is not checked, so it is the programmer's responsibility to ensure they are all valid.
<i>subject</i>	A null-terminated string containing the subject line.
<i>body</i>	A null-terminated string containing the text of the email to send. This must be character encoding "ISO-8859-1 (latin1)" (the ECL default character set). Text in any other character set must be sent as an <i>attachment</i> .
<i>attachment</i>	A DATA value containing the binary data to attach.
<i>mimetype</i>	A null-terminated string containing the MIME-type of the <i>attachment</i> , which may include entrymeters (such as 'text/plain; charset=ISO-8859-3'). When attaching general binary data for which no specific MIME type exists, use 'application/octet-stream'.
<i>filename</i>	A null-terminated string containing the name of the <i>attachment</i> for the mail reader to display.
<i>server</i>	Optional. A null-terminated string containing the name of the mail server. If omitted, defaults to the value in the SMTPserver environment variable.
<i>port</i>	Optional. An UNSIGNED4 integer value containing the port number. If omitted, defaults to the value in the SMTPport environment variable.
<i>sender</i>	Optional. A null-terminated string containing the address of the sender. If omitted, defaults to the value in the emailSenderAddress environment variable.

The **SendEmailAttachData** function sends an email message with a binary *attachment*.

Example:

```
DATA15 attachment := D'test attachment';
STD.System.Email.SendEmailAttachData( 'me@mydomain.com',
                                     'testing 1,2,3',
                                     'this is a test message',
                                     attachment,
                                     'application/octet-stream',
                                     'attachment.txt');
```

SendEmailAttachText

STD.System.Email.SendEmailAttachText (*sendto*, *subject*, *body*, *attachment*, *mimietype*, *filename*, *server*, *port*, *sender*)

<i>sendto</i>	A null-terminated string containing a comma-delimited list of the addresses of the intended recipients. The validity of the addresses is not checked, so it is the programmer's responsibility to ensure they are all valid.
<i>subject</i>	A null-terminated string containing the subject line.
<i>body</i>	A null-terminated string containing the text of the email to send. This must be character encoding "ISO-8859-1 (latin1)" (the ECL default character set). Text in any other character set must be sent as an <i>attachment</i> .
<i>attachment</i>	A null-terminated string containing the text to attach.
<i>mimetype</i>	A null-terminated string containing the MIME-type of the <i>attachment</i> , which may include entrymeters (such as 'text/plain; charset=ISO-8859-3').
<i>filename</i>	A null-terminated string containing the name of the <i>attachment</i> for the mail reader to display.
<i>server</i>	Optional. A null-terminated string containing the name of the mail server. If omitted, defaults to the value in the SMTPserver environment variable.
<i>port</i>	Optional. An UNSIGNED4 integer value containing the port number. If omitted, defaults to the value in the SMTPport environment variable.
<i>sender</i>	Optional. A null-terminated string containing the address of the sender. If omitted, defaults to the value in the emailSenderAddress environment variable.

The **SendEmailAttachText** function sends an email message with a text *attachment*.

Example:

```
STD.System.Email.SendEmailAttachText( 'me@mydomain.com', 'testing 1,2,3',  
                                       'this is a test message', 'this is a test attachment',  
                                       'text/plain; charset=ISO-8859-3', 'attachment.txt');
```

Workunit Services

WorkunitExists

STD.System.Workunit.WorkunitExists(*wuid* [, *online*] [, *archived*])

<i>wuid</i>	A null-terminated string containing the WorkUnit IDentifier to locate.
<i>online</i>	Optional. A Boolean true/false value specifying whether the search is performed online. If omitted, the default is TRUE.
<i>archived</i>	Optional. A Boolean true/false value specifying whether the search is performed in the archives. If omitted, the default is FALSE.
Return:	WorkunitExists returns a BOOLEAN value.

The **WorkunitExists** function returns whether the *wuid* exists.

Example:

```
OUTPUT(STD.System.Workunit.WorkunitExists('W20070308-164946'));
```

WorkunitList

STD.System.Workunit.WorkunitList (*lowwuid* [, *highwuid*] [, *username*] [, *cluster*] [, *jobname*] [, *state*] [, *priority*] [, *fileread*] [, *filewritten*] [, *roxiecluster*] [, *eclcontains*] [, *online*] [, *archived*] [, *appvalues*])

<i>lowwuid</i>	A null-terminated string containing the lowest WorkUnit IDentifier to list. This may be an empty string.
<i>highwuid</i>	Optional. A null-terminated string containing the highest WorkUnit IDentifier to list. If omitted, the default is an empty string.
<i>cluster</i>	Optional. A null-terminated string containing the name of the cluster the workunit ran on. If omitted, the default is an empty string.
<i>jobname</i>	Optional. A null-terminated string containing the name of the workunit. This may contain wildcard (* ?) characters. If omitted, the default is an empty string.
<i>state</i>	Optional. A null-terminated string containing the state of the workunit. If omitted, the default is an empty string.
<i>priority</i>	Optional. A null-terminated string containing the priority of the workunit. If omitted, the default is an empty string.
<i>fileread</i>	Optional. A null-terminated string containing the name of a file read by the workunit. This may contain wildcard (* ?) characters. If omitted, the default is an empty string.
<i>filewritten</i>	Optional. A null-terminated string containing the name of a file written by the workunit. This may contain wildcard (* ?) characters. If omitted, the default is an empty string.
<i>roxiecluster</i>	Optional. A null-terminated string containing the name of the Roxie cluster. If omitted, the default is an empty string.
<i>eclcontains</i>	Optional. A null-terminated string containing text to search for in the workunit's ECL code. This may contain wildcard (* ?) characters. If omitted, the default is an empty string.
<i>online</i>	Optional. A Boolean true/false value specifying whether the search is performed online. If omitted, the default is TRUE.
<i>archived</i>	Optional. A Boolean true/false value specifying whether the search is performed in the archives. If omitted, the default is FALSE.
<i>appvalues</i>	Optional. A null-terminated string containing application values to search for. Use a string of the form appname/key=value or appname/*=value.
Return:	WorkunitList returns a DATASET.

The **WorkunitList** function returns a dataset of all workunits that meet the search criteria specified by the parameters passed to the function. All the parameters are search values and all but the first are omissible, therefore the easiest way to pass a particular single search parameter would be to use the NAMED parameter passing technique.

The resulting DATASET is in this format:

```
WorkunitRecord := RECORD
  STRING24 wuid;
  STRING owner{MAXLENGTH(64)};
  STRING cluster{MAXLENGTH(64)};
  STRING roxiecluster{MAXLENGTH(64)};
  STRING job{MAXLENGTH(256)};
  STRING10 state;
  STRING7 priority;
  STRING20 created;
  STRING20 modified;
  BOOLEAN online;
```

```
    BOOLEAN protected;  
END;
```

Example:

```
OUTPUT(STD.System.Workunit.WorkunitList(''));  
//list all current workunits  
OUTPUT(STD.System.Workunit.WorkunitList('',  
    NAMED eclcontains := 'COUNT'));  
//list only those where the ECL code contains the word 'COUNT'  
//this search is case insensitive and does include comments  
  
STD.System.Workunit.SetWorkunitAppValue('MyApp','FirstName','Jim',TRUE);  
OUTPUT(STD.System.Workunit.WorkunitList(appvalues := 'MyApp/FirstName='Jim'));  
//returns a list of workunits with app values where FirstName='Jim'
```

See Also: SetWorkunitAppValue

SetWorkunitAppValue

STD.System.Workunit.SetWorkunitAppValue (*app*, *key*, *value*, [*overwrite*])

<i>app</i>	The application name to set.
<i>key</i>	The name of the value to set.
<i>value</i>	The value to set.
<i>overwrite</i>	A boolean TRUE or FALSE flag indicating whether to allow the value to overwrite an existing value. Default is TRUE..
Return:	SetWorkunitAppValue returns TRUE if the value was set successfully.

The **SetWorkunitAppValue** function sets an application value in the current workunit. It returns TRUE if the value was set successfully.

Example:

```
IMPORT STD;
STD.System.Workunit.SetWorkunitAppValue('MyApp', 'FirstName', 'Jim', TRUE);
OUTPUT(STD.System.Workunit.WorkunitList(appvalues := 'MyApp/FirstName='Jim'));
//returns a list of workunits with app values where FirstName='Jim'
```

See Also: WorkunitList

WUIDonDate

STD.System.Workunit.WUIDonDate (*year, month, day, hour, minute*)

<i>year</i>	An unsigned integer containing the year value.
<i>month</i>	An unsigned integer containing the month value.
<i>day</i>	An unsigned integer containing the day value.
<i>hour</i>	An unsigned integer containing the hour value.
<i>minute</i>	An unsigned integer containing the minute value.
Return:	WUIDonDate returns a VARSTRING value.

The **WUIDonDate** function returns a valid WorkUnit IDentifier for a workunit that meets the passed parameters.

Example:

```
lowwuid := STD.System.Workunit.WUIDonDate(2008,02,13,13,00);  
highwuid := STD.System.Workunit.WUIDonDate(2008,02,13,14,00);  
OUTPUT(STD.System.Workunit.WorkunitList(lowwuid,highwuid));  
//returns a list of workunits between 1 & 2 PM on 2/13/08
```

WUIDdaysAgo

STD.System.Workunit.WUIDdaysAgo (*daysago*)

<i>daysago</i>	An unsigned integer containing the number of days to go back.
Return:	WUIDdaysAgo returns a VARSTRING value.

The **WUIDdaysAgo** function returns a valid WorkUnit IDentifier for a workunit that would have run within the last *daysago* days.

Example:

```
daysago := STD.System.Workunit.WUIDdaysAgo(3);  
OUTPUT(STD.System.Workunit.WorkunitList(daysago));  
//returns a list of workunits run in the last 72 hours
```

WorkunitTimeStamps

STD.System.Workunit.WorkunitTimeStamps (*wuid*)

<i>wuid</i>	A null-terminated string containing the WorkUnit Identifier.
Return:	WorkunitTimeStamps returns a DATASET value.

The **WorkunitTimeStamps** function returns a DATASET with this format:

```
EXPORT WsTimeStamp := RECORD
  STRING32 application;
  STRING16 id;
  STRING20 time;
  STRING16 instance;
END;
```

Each record in the returned dataset specifies a step in the workunit's execution process (creation, compilation, etc.).

Example:

```
OUTPUT(STD.System.Workunit.WorkunitTimeStamps('W20070308-164946'));
/* produces output like this:
'workunit      ','Created ','2008-02-13T18:28:20Z',' '
'workunit      ','Modified','2008-02-13T18:32:47Z',' '
'EclServer     ','Compiled','2008-02-13T18:28:20Z','10.173.9.2:0 '
'EclAgent      ','Started ','2008-02-13T18:32:35Z','training009003'
'Thor - graph1','Finished','2008-02-13T18:32:47Z','training009004'
'Thor - graph1','Started ','2008-02-13T18:32:13Z','training009004'
'EclAgent      ','Finished','2008-02-13T18:33:09Z','training009003'
*/
```

WorkunitMessages

STD.System.Workunit.WorkunitMessages (*wuid*)

<i>wuid</i>	A null-terminated string containing the WorkUnit IDentifier.
Return:	WorkunitMessages returns a DATASET value.

The **WorkunitMessages** function returns a DATASET with this format:

```
EXPORT WsMessage := RECORD
  UNSIGNED4 severity;
  INTEGER4 code;
  STRING32 location;
  UNSIGNED4 row;
  UNSIGNED4 col;
  STRING16 source;
  STRING20 time;
  STRING message{MAXLENGTH(1024)};
END;
```

Each record in the returned dataset specifies a message in the workunit.

Example:

```
OUTPUT(STD.System.Workunit.WorkunitMessages('W20070308-164946'));
```


WorkunitFilesRead

STD.System.Workunit.WorkunitFilesRead (*wuid*)

<i>wuid</i>	A null-terminated string containing the WorkUnit Identifier.
Return:	WorkunitFilesRead returns a DATASET value.

The **WorkunitFilesRead** function returns a DATASET with this format:

```
EXPORT WSFileRead := RECORD
  STRING name{MAXLENGTH(256)};
  STRING cluster{MAXLENGTH(64)};
  BOOLEAN isSuper;
  UNSIGNED4 usage;
END;
```

Each record in the returned dataset specifies a file read by the workunit.

Example:

```
OUTPUT(STD.System.Workunit.WorkunitFilesRead('W20070308-164946'));
/* produces results that look like this
'rttest::difftest::superfile','thor','true','1'
'rttest::difftest::base1','thor','false','1'
*/
```

WorkunitFilesWritten

STD.System.Workunit.WorkunitFilesWritten (*wuid*)

<i>wuid</i>	A null-terminated string containing the WorkUnit Identifier.
Return:	WorkunitFilesWritten returns a DATASET value.

The **WorkunitFilesWritten** function returns a DATASET with this format:

```
EXPORT wsFileRead := RECORD
  STRING name{MAXLENGTH(256)};
  STRING10 graph;
  STRING cluster{MAXLENGTH(64)};
  UNSIGNED4 kind;
END;
```

Each record in the returned dataset specifies a file written by the workunit.

Example:

```
OUTPUT(STD.System.Workunit.WorkunitFilesWritten('W20070308-164946'));
/* produces results that look like this
'rttest::testfetch','graph1','thor','0'
*/
```

WorkunitTimings

STD.System.Workunit.WorkunitTimings (*wuid*)

<i>wuid</i>	A null-terminated string containing the WorkUnit Identifier.
Return:	WorkunitTimings returns a DATASET value.

The **WorkunitTimings** function returns a DATASET with this format:

```
EXPORT WSTiming := RECORD
  UNSIGNED4 count;
  UNSIGNED4 duration;
  UNSIGNED4 max;
  STRING name{MAXLENGTH(64)};
END;
```

Each record in the returned dataset specifies a timing for the workunit.

Example:

```
OUTPUT(STD.System.Workunit.WorkunitTimings('W20070308-164946'));
/* produces results that look like this
'1','4','4','EclServer: tree transform'
'1','0','0','EclServer: tree transform: normalize.scope'
'1','1','1','EclServer: tree transform: normalize.initial'
'1','18','18','EclServer: write c++'
'1','40','40','EclServer: generate code'
'1','1010','1010','EclServer: compile code'
'1','33288','33288','Graph graph1 - 1 (1)'
'1','33629','33629','Total thor time: '
'2','1','698000','WorkUnit_lockRemote'
'1','2','2679000','SDS_Initialize'
'1','0','439000','Environment_Initialize'
'1','33775','3710788928','Process'
'1','1','1942000','WorkUnit_unlockRemote'
*/
```

BLAS Support

This section provides support for Basic Linear Algebra Subprogram support.

The BLAS functions use the column major mapping for the storage of a matrix. This is the mapping used in Fortran, and has the entries of the first column followed by the entries of the second column. This is the transpose of the row major form commonly used in the C language where the entries of the first row are followed by the entries of the second row.

Types

STD.BLAS.Types

<i>value_t</i>	REAL8
<i>dimension_t</i>	UNSIGNED4
<i>matrix_t</i>	SET OF REAL8
<i>Triangle</i>	ENUM(UNSIGNED1, Upper=1, Lower=2)
<i>Diagonal</i>	ENUM(UNSIGNED1, UnitTri=1, NotUnitTri=2)
<i>Side</i>	ENUM(UNSIGNED1, Ax=1, xA=2)

Types for the Block Basic Linear Algebra Sub-programs support

ICellFunc

STD.BLAS.ICellFunc(*v*, *r*, *c*);

<i>v</i>	The value
<i>r</i>	The row ordinal
<i>c</i>	The column ordinal
Return:	The updated value

ICellFunc is the function prototype for Apply2Cells.

Example:

```
IMPORT STD;
REAL8 my_func(STD.BLAS.Types.value_t v, STD.BLAS.Types.dimension_t x, STD.BLAS.Types.dimension_t y)
    := 1/v; //set element to the reciprocal value
```

See Also: Apply2Cells

Apply2Cells

STD.BLAS.Apply2Cells(*m*, *n*, *x*, *f*);

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>x</i>	Matrix
<i>f</i>	Function to apply
Return:	The updated matrix

The **Apply2Cells** function iterates a matrix and applies a function to each cell.

Example:

```
IMPORT STD;
STD.BLAS.Types.value_t example_1(STD.BLAS.Types.value_t v,
                                STD.BLAS.Types.dimension_t x,
                                STD.BLAS.Types.dimension_t y) := FUNCTION
    RETURN IF(x=y, 1.0, 1/v);
END;

init_mat := [1, 2, 4, 4, 5, 10, 2, 5, 2];
new_mat := STD.BLAS.Apply2Cells(3, 3, init_mat, example_1);

// The new_mat matrix will be [1, .5, .25, .25, 1, .1, .5, .2, 1]
```

See Also: ICellFunc

dasum

STD.BLAS.dasum(*m*, *x*, *incx*, *skipped*);

<i>m</i>	Number of entries
<i>x</i>	The column major matrix holding the vector
<i>incxx</i>	The increment for <i>x</i> , 1 in the case of an actual vector
<i>skipped</i>	The number of entries stepped over. Default is zero.
Return:	The sum of the absolute values

The **dasum** function gets the absolute sum, the 1 norm of a vector.

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t test_data := [2, -2, -3, 3, 1, 3, -1, -1, 1];
STD.BLAS.dasum(9, test_data, 1); //sums the absolute values of the matrix, and returns 17
```


daxpy

STD.BLAS.daxpy(*N*, *alpha*, *X*, *incX*, *Y*, *incY*, *x_skipped*, *y_skipped*);

<i>N</i>	Number of entries
<i>alpha</i>	The column major matrix holding the vector
<i>X</i>	The increment for x, 1 in the case of an actual vector
<i>incX</i>	The column major matrix holding the vector X
<i>Y</i>	The column major matrix holding the vector Y
<i>incY</i>	The increment or stride of Y
<i>x_skipped</i>	The number of entries stepped over. to get to the first X .
<i>y_skipped</i>	The number of entries stepped over. to get to the first Y .
Return:	The updated matrix

The **daxpy** function is used to sum two vectors or matrices with a scalar multiplier applied during the sum operation..

Example:

```
IMPORT STD;
STD.BLAS.Types.t_matrix term_1 := [1, 2, 3];
STD.BLAS.Types.t_matrix term_2 := [3, 2, 1].
STD.BLAS.daxpy(3, 2, term_1, 1, term_2, 1); // result is [5, 6, 7]
```

dgemm

STD.BLAS.dgemm(*transposeA*, *transposeB*, *M*, *N*, *K*, *alpha*, *A*, *B*, *beta*, *C*);

<i>transposeA</i>	True when transpose of A is used
<i>transposeB</i>	True when transpose of B is used
<i>M</i>	Number of rows in product
<i>N</i>	Number of columns in product
<i>K</i>	Number of columns/rows for the multiplier/multiplicand
<i>alpha</i>	Scalar used on A
<i>A</i>	Matrix A
<i>B</i>	Matrix B
<i>beta</i>	Scalar for matrix C
<i>C</i>	Matrix C (or empty)
Return:	The updated matrix

The **dgemm** function is used to multiply two matrices and optionally add that product to another matrix.

Example:

```
IMPORT STD;
STD.BLAS.Types.t_matrix term_a := [2, 4, 8];
STD.BLAS.Types.t_matrix term_c := [2, 1, 1];

STD.BLAS.dgemm(TRUE, FALSE, 3, 3, 1, 1, term_a, term_b);
    //the outer product of the term_a and term_b vectors
    //result is [4,8, 16, 2, 4, 8, 2, 4, 8]
```

dgetf2

STD.BLAS.dgetf2(*m*, *n*, *a*);

<i>m</i>	Number of rows of matrix a
<i>n</i>	Number of columns of matrix a
<i>a</i>	Matrix a
Return:	Composite matrix of factors, lower triangle has an implied diagonal of ones. Upper triangle has the diagonal of the composite.

The **dgetf2** function produces a combine lower and upper triangular factorization.

Example:

```
IMPORT STD;  
STD.BLAS.Types.t_matrix test := [2,4,6,3,10,25, 9,34,100];  
STD.BLAS.dgetf2(3, 3, test); //result is [2,2,3,3,4,4,9,16,25];
```

dpotf2

STD.BLAS.dpotf2(*tri*., *r*, *A*, *clear*);

<i>tri</i>	Indicates whether upper or lower triangle is used
<i>r</i>	Number of rows/columns in the square matrix
<i>A</i>	The square matrix A
<i>clear</i>	Clears the unused triangle
Return:	The triangular matrix requested

The **dpotf2** function computes the Cholesky factorization of a real symmetric positive definite matrix A. The factorization has the form $A = U^{**T} * U$ if the *tri* parameter is `Triangle.Upper`, or $A = L * L^{**T}$ if the *tri* parameter is `Triangle.Lower`. This is the unblocked version of the algorithm, calling Level 2 BLAS.

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t symmetric_pos_def := [4, 6, 8, 6, 13, 18, 8, 18, 29];
Lower_Triangle := BLAS.dpotf2(STD.BLAS.Types.Triangle.lower, 3, symmetric_pos_def);
```

dscal

STD.BLAS.dscal(*N*, *alpha*, *X*, *incX*, *skipped*);

<i>N</i>	Number of elements in the vector
<i>alpha</i>	The scaling factor
<i>X</i>	The column major matrix holding the vector
<i>incX</i>	The stride to get to the next element in the vector
<i>skipped</i>	The number of elements skipped to get to the first element
Return:	The updated matrix

The **dscal** function scales a vector *alpha*.

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t test := [1, 1, 1, 2, 2, 2, 3, 3, 3];
result := STD.BLAS.dscal(9, 2.0, test, 1); // multiply each element by 2
```

dsyrk

STD.BLAS.dsyrk(*tri*, *transposeA*, *N*, *K*, *alpha*, *A*, *beta*, *C*, *clear*);

<i>tri</i>	Indicates whether upper or lower triangle is used
<i>transposeA</i>	Transpose the A matrix to be NxK
<i>N</i>	Number of rows
<i>K</i>	Number of columns in the update matrix or transpose
<i>alpha</i>	The alpha scalar
<i>A</i>	The update matrix, either NxK or KxN
<i>beta</i>	The beta scalar
<i>C</i>	The matrix to update
<i>clear</i>	Clear the triangle that is not updated. BLAS assumes that symmetric matrices have only one of the triangles and this option lets you make that true.
Return:	The updated matrix

The **dsyrk** function implements a symmetric rank update $C \leftarrow \alpha A A^T + \beta C$ or $c \leftarrow \alpha A A^{**T} + \beta C$. C is N x N.

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t initC := [1, 1, 1, 2, 2, 2, 3, 3, 3];
STD.BLAS.Types.matrix_t initA := [1, 1, 1];
Test1_mat := STD.BLAS.dsyrk(STD.BLAS.Types.Triangle.upper, FALSE, 3, 1, 1, initA, 1, initC, TRUE)
```

dtrsm

STD.BLAS.dtrsm(*side*, *tri*, *transposeA*, *diag*, *M*, *N*, *lda*, *alpha*, *A*, *B*);

<i>side</i>	Side for A, Side.Ax is $\text{op}(A) X = \alpha B$
<i>tri</i>	Indicates whether upper or lower triangle is used
<i>transposeA</i>	Is $\text{op}(A)$ the transpose of A
<i>diag</i>	The diagonal (an implied unit diagonal or supplied)
<i>M</i>	Number of rows
<i>N</i>	Number of columns
<i>lda</i>	The leading dimension of the A matrix, either M or N
<i>alpha</i>	The scalar multiplier for B
<i>A</i>	A triangular matrix
<i>B</i>	The matrix of values for the solve
Return:	The matrix of coefficients to get B

The **dtrsm** function is a triangular matrix solver. $\text{op}(A) X = \alpha B$ or $X \text{op}(A) = \alpha B$ * where op is Transpose, X and B is $M \times N$

Example:

```
IMPORT STD;
Side := STD.BLAS.Types.Side;
Diagonal := STD.BLAS.Types.Diagonal;
Triangle := STD.BLAS.Types.Triangle;
STD.BLAS.Types.matrix_t left_a0 := [2, 3, 4, 0, 2, 3, 0, 0, 2];
STD.BLAS.Types.matrix_t mat_b := [4, 6, 8, 6, 13, 18, 8, 18, 29];

Test1_mat := STD.BLAS.dtrsm(Side.Ax, Triangle.Lower, FALSE, Diagonal.NotUnitTri,
                           3, 3, 3, 1.0, left_a0, mat_b);
```

extract diag

STD.BLAS.extract_diag (*m.n.x*);

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>x</i>	The matrix from which to extract the diagonal
Return:	Diagonal matrix

The **extract_diag** function extracts the diagonal of the matrix

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t x := [1.0, 2.0, 3.0, 2.0, 2.0, 2.0, 4.0, 4.0, 4.0];
diagonal_only := STD.BLAS.extract_diag(3, 3, x);
```


extract_tri

STD.BLAS.extract_tri (*m*, *n*, *tri*, *dt*, *a*);

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>tri</i>	Indicates whether upper or lower triangle is used
<i>dt</i>	Use Diagonal.NotUnitTri or Diagonal.UnitTri
<i>a</i>	The matrix, usually a composite from factoring
Return:	Triangle

The **extract_tri** function extracts the upper or lower triangle. The diagonal can be the actual or implied unit diagonal.

Example:

```
IMPORT STD;
Diagonal := STD.BLAS.Types.Diagonal;
Triangle := STD.BLAS.Types.Triangle;
STD.BLAS.Types.matrix_t x := [1.0, 2.0, 3.0, 2.0, 2.0, 2.0, 4.0, 4.0, 4.0];
triangle := STD.BLAS.extract_tri(3, 3, Triangle.upper, Diagonal.NotUnitTri, x);
```

make diag

STD.BLAS.make_diag (*m*, *v*, *X*);

<i>m</i>	Number of diagonal entries
<i>v</i>	Option value, default is 1
<i>X</i>	Optional input of diagonal values, multiplied by <i>v</i>
Return:	A diagonal matrix

The **make_diag** function generates a diagonal matrix.

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t init1 := [1.0, 2.0, 3.0, 4.0];
Square := STD.BLAS.make_diag(4, 1, init1); //4x4 with diagonal 1, 2, 3, 4
```

make_vector

STD.BLAS.make_vector (m , v);

m	Number of elements
v	The values, default is 1
Return:	The vector

The **make_vector** function generates a vector of dimension n

Example:

```
IMPORT STD;  
twos_vector := STD.BLAS.make_vector(4, 2); // a vector of [2, 2, 2, 2]
```

trace

STD.BLAS.trace (m , n , x);

m	Number of rows
n	Number of columns
x	The matrix
Return:	The trace (sum of the diagonal entries)

The **trace** function computes the trace of the input matrix

Example:

```
IMPORT STD;
STD.BLAS.Types.matrix_t x := [1.0, 2.0, 3.0, 2.0, 2.0, 2.0, 4.0, 4.0, 4.0];
trace_of_x := STD.BLAS.trace(3,3,x); // the trace is 7
```