KREO HMI

Structured Text Script guideline

Software

EXTERNAL REFERENCES

Sources referenced in the present document:

[1] ST Scripts Definition
R:\WCE\Documenti\BlueOcean\RT7\Scripts\ST\ST-Scripts Definition.doc
Description of implemented ST scripts syntax, functionalities and limitations

[2] ST Scripts P-CODE
R:\WCE\Documenti\BlueOcean\RT7\Scripts\ST\ST-Scripts P-Code.doc
Specification of formats and mechanics involved in scripts P-Code generation and interpretation

HISTORY

0.1 - 25.01.2019 - First document release

1.0 - 09.04.2019 - First distributed version
Implementation limited to rt-only methods:
missing SoftPlc, EveryWare, UI and Printing methods

1.1 - 02.12.2019 - Added tools for structured tags management
see *TAG_GETFIELDOFFSET* and *TAG_GETFIELDADDRESS*

1.2 - 14.01.2020 - Added descriptions for ST standard functions
22.01.2020 - Extended File functions with support of $SECTIONS in paths

1.3 - 13.02.2020 - *LIBRARY* functions section isolated and expanded
COMLIB and *COMVAR* sections still missing details
31.03.2020 - Added "variant" access support to *TAG_READITEM* and *TAG_WRITEITEM*

1.4 - 13.05.2020 - Included basic functions for matrix-users (to be integrated after "phase 2")

1.5 - 25.05.2020 - Extended functions for revised FDA features
see *USER_ISLOCKED*, *USER_PERMANENTLOCK* for permanent lock states
see *USER_ADD*, *USER_SETSIGNATURE* for electronic signature support
see *AUDIT_READ* for new logged information
see *AUDIT_EXPORT*, *AUDIT_RESET* for exports extensions

1.6 - 01.07.2020 - Added function *SAVESCREEN*
Added function *GETCLIENTID*

1.7 - 15.07.2020 - Added the whole section for EveryWare functions (both basic and mailing-oriented)
see chapter *10. Common - EXTERNAL*

1.8 - 23.07.2020 - Added the whole section for PDF creation functions
see chapter *9. Common - PRINT*
31.07.2020 - Extended users' export (see updated parameters in *USER_EXPORT*)

1.9 - 21.09.2020 - Added specifications about the - already released - *COMLIB* functions

1.10 - 23.09.2020 - Added specifications about the - already released - *COMVAR* functions

1.11 - 15.10.2020 - Added function *RECIPE_COMPARESET*
added functions *RECIPE_GETFIELDVALUE* and *RECIPE_SETFIELDVALUE*
added recipe name parameter to *RECIPE_UPLOAD*

1.12 - 11.11.2020 - Added parameters to function *RUNAPPLICATION*

1.13 - 18.11.2020 - Added function *SPLIT* for strings management
Added functions *RECIPE_GETFIELDSNUMBER* and *RECIPE_GETFIELDNAME*
Added variables *RT_WORKMODE* and *RT_SIMULATION*

1.14 - 04.12.2020 - Added function *RECIPE_GETFIELDINDEX*

1.15 - 14.01.2021 - Added functions *ALARM_GETIDFROMKEY* and *ALARM_GETMSGFROMKEY*
Extended function *RECIPE_GETFIELDNAME* (multiple identification modes)
Extended function *RECIPE_GETTAGNAME* (support implicit comparison tags)

Extended function *RECIPE_COMPARESET* (support implicit comparison tags)
Added function *RECIPE_GETCOMPAREINDEX*
Added function *RECIPE_COMPAREFIELD*

1.16 - 19.02.2021 - Added function *FILE_ABSOLUTEPATH*
Added functions *DLOG_GETDISCARDINVALID* and *DLOG_SETDISCARDINVALID*
Added functions for buzzer and lamp support

1.17 - 17.03.2021 - Added functions *ALARM_GETINFO* and *ALARM_GETNAMEFROMKEY*

1.18 - 10.09.2021 - Extended function *USER_ADD* with RFID parameters
Added function *USER_SETRFID*
Added function *USER_HASRFID*
Added functions *TAG_SETVALUE*, *TAG_FLUSHVALUE*
Added functions *TAG_GETFIELDVALUE*, *TAG_SETFIELDVALUE*

1.19 - 13.10.2021 - Extended recipe functions with new parameter for event logging inhibition
(see *RECIPE_LOAD*, *RECIPE_SAVE*, *RECIPE_UPLOAD*, *RECIPE_DOWNLOAD*, *RECIPE_UPLOADBUFFER*, *RECIPE_DOWNLOADBUFFER*, *RECIPE_DELETE*, *RECIPE_RENAME*)

1.20 - 20.10.2021 - Added function *SHOWTASKBAR*

1.21 - 23.11.2021 - Extended recipes import/export functions with the ability to select the needed recipes
(see *RECIPE_EXPORT*, *RECIPE_EXPORTFLAT*, *RECIPE_IMPORT*)

1.22 - 02.02.2022 - Added functions *GET/SETCLIENTDEBUGRIGHTS* and *GETCLIENTDEBUGSTATE*
Added functions *GET/SETCLIENTOFFSCAN*

1.23 - 25.03.2022 - Marks the integration of *_RETURNCODE* and *_RETURNERROR* statements (specified in [1])

1.24 - 08.04.2022 - Added functions for recorded alarms information browsing
(see *ALARM_GETFIRST/NEXTxxx* functions and *ALARM_RECxxx* variables)
Extended *PDF_OPEN* with protection parameters
Added function *SIZEOF*

1.25 - 03.05.2022 - Added function *TAG_GETCLIENTTAGNAME*
Added functions *SET/GETCLIENTKEYBURST*

1.26 - 10.05.2022 - Added function *TYPEOF*
Added type conversion aliases (*C$$$*) for all the functions *ANY_TO_$$$*
Added function *TAG_ASSIGNSTRUCT*
Fixed output type of function *TAG_GETCLIENTTAGNAME*

1.27 - 31.05.2022 - Extended *xTRIM* functions
Added functions *ALARM_EXPORTCONFIG* and *DLOG_EXPORTCONFIG*

1.28 - 24.06.2022 - Added functions *SETRESTAPIPREFIX* and *SETRESTAPIRESPONSE*

1.29 - 29.06.2022 - Added functions *RECIPE_GETSTRRECORD* and *RECIPE_GETSTRRECORDS*
Added functions *ALARM_GETFIRSTHPACK* and *ALARM_GETNEXTHPACK*

1.30 - 13.09.2022 - Extended function *AUDIT_READ* (updated information set for clients and tags)
04.10.2022 - Added function *RECIPE_SETFIELDEXPORT*

1.31 - 26.05.2023 - Added function *SETHHLEDSTATE*

1.32 - 30.06.2023 - Added function *REPORT_EXPORT*

1.33 - 20.09.2023 - Extended and documented *TRACE* function

1.34 - 03.10.2023 - Extended *REPORT_EXPORT* function with language parameter

1.35 - 07.02.2024 - Added functions *SENDRESTAPIREQUEST* and *GETRESTAPIRESPONSE*
See also *RESTAPIRESPONSE* variable and *RESTxxx* constants

1.36 - 19.03.2024 - Added parameter *TRACE* to function *SENDRESTAPIREQUEST*
Added function *REPORT_ENABLESECTION*
See also the related *REPORT_CLASSxxx* constants

1.37 - 09.04.2024 - Added USERS selection to function *REPORT_ENABLESECTION*
(see *REPORTCLASSUSERS* among constants as well)
Extended functionality of function *REPORT_ENABLESECTION*
Added parameters FROM/TO to function *DLOG_EXPORT*
Added parameters FROM/TO to function *ALARM_EXPORTHISTORY*

1.38 - 02.05.2024 - Added function *REFRESHRTC*
Added a set of variables (*TIMExxx*) with time-related information

[–› actBase.cpp / ModuleVersion.inc]

TABLE OF CONTENTS

1. DOCUMENT SCOPE

Purpose of the present document is to give a complete list of all the functions implemented in the ST scripting language supported by ESA platforms.
The document will provide a short reference and a list of correspondences between the new ST functions and the 'old' VBS functions available on EW platform, along with a detailed description of each one.

2. FUNCTIONS LIST

The following is the full list of the implemented functions.
Along with the reference itself, the most likely match with a VBS function is indicated.
Not all of the functions are ESA extensions: some of them are simply part of the basic language environment (gathered in the initial part of the list).
Note that many of the ST functions has no equal in the VBS environment; similarly, some of the VBS functions might make no sense in ST. The principle goes for both ESA functions and native language functions.

Notations:
- the ST functions with no match in VBS are left as empty [/] cells;
- matching standard VBS functions are specified in [purple];
- matching ESA VBS functions are given in [black];
- unmatching but similar VBS functions could be pointed out as [side notes];
- VBS functions meant for UI access are given in [orange];
  these functions will not be implemented in ST as per MKT specifications.

ST function	VBS correspondence	Description
ST STANDARDS		
Mathematics		
ADD (A,B,C,...)	/ (operator only)	sum of all the parameters A+B+C+...
SUB (A,B)	/ (operator only)	subtraction A-B
MUL (A,B,C,...)	/ (operator only)	multiplication of all the parameters A*B*C*...
DIV (A,B)	/ (operator only)	division A/B
MOD (A,B)	/ (operator only)	remainder or fractional part of A/B
ABS (A)	Abs	absolute value of A
SQRT (A)	Sqr	square root of A
LN (A)	Log	natural logarithm of A
LOG (A)	/	base 10 logarithm of A
EXP (A)	Exp	e**A (where 'e' is the natural number)
POW (A,B)	/	A**B
SIN (A)	Sin	sine of A
COS (A)	Cos	cosine of A
TAN (A)	Tan	tangent of A
ASIN (A)	/	arc-sine of A
ACOS (A)	/	arc-cosine of A
ATAN (A)	Atn	arc-tangent of A
FRACTION (A)	/	(not standard ST) get the fractional part of A
TRUNC (A)	/ (similar: Int, Fix)	(not standard ST) conv. A to int.; truncate decimals
ROUND (A)	Round	(not standard ST) conv. A to int.; round decimals
MIN (A,B,C,...)	/	minimum of A,B,C,...
MAX (A,B,C,...)	/	maximum of A,B,C,...
LIMIT (N,V,X)	/	limit the value V between a min N and a max X
SEL (C,F,T)	/	return F if the condition C is FALSE; T if TRUE
MUX (N,A,B,C,...)	/	return the Nth (+1) parameter value
RND ()	Rnd	(not standard ST) obtain a pseudo-random value
Bits		
SHL (V,B)	/	shift value V left of B bits
SHR (V,B)	/	shift value V right of B bits
ROL (V,B)	/	rotate value V left of B bits
ROR (V,B)	/	rotate value V right of B bits
AND (A,B,C,...)	/ (operator only)	logical/bitwise AND of all inputs (A,B,C,...)
OR (A,B,C,...)	/ (operator only)	logical/bitwise OR of all inputs (A,B,C,...)
XOR (A,B,C,...)	/ (operator only)	logical/bitwise XOR of all inputs (A,B,C,...)
NOT (A)	/ (operator only)	logical/bitwise NOT of A
Comparison		

LT (A,B,C,...)	/ (operator only)	compare A<B<C<...
LE (A,B,C,...)	/ (operator only)	compare A≤B≤C≤...
GT (A,B,C,...)	/ (operator only)	compare A>B>C>...
GE (A,B,C,...)	/ (operator only)	compare A≥B≥C≥...
EQ (A,B,C,...)	/ (operator only)	compare A=B=C=... TRUE if all values are equal
NE (A,B,C,...)	/ (operator only)	compare A,B,C,... TRUE if all values are different
String		
LEN (S)	Len	return the length of string S
LEFT (S,L)	Left	return the left L characters of string S
RIGHT (S,L)	Right	return the right L characters of string S
MID (S,L,P)	Mid	return L characters starting at P of string S
CONCAT (S1,S2,S3,...)	/	return strings S1,S2,S3,... joined together
INSERT (S1,S2,P)	/	insert string S2 into S1 at position P
DELETE (S,L,P)	/	delete L characters from position P in string S
REPLACE (S1,S2,L,P)	Replace	replace L chars from pos P in string S1 with S2
REVERSE (S)	StrReverse	(not standard ST) reverse a string
SPLIT (S,[C],[L],[P])	Split	(not standard ST) split a string in an array of pieces
FIND (S1,S2)	InStr	find the start of string S2 in string S1
RFIND (S1,S2)	InStrRev	(not standard ST) find S2 in S1; search from right
LCASE (S)	LCase	(not standard ST) convert the string S in lower case
UCASE (S)	UCase	(not standard ST) convert the string S in upper case
TRIM (S,C)	Trim	(not standard ST) remove chars C around a string
LTRIM (S,C)	LTrim	(not standard ST) remove chars C on the left
RTRIM (S,C)	RTrim	(not standard ST) remove chars C on the right
SETLENGTH (S,C,L,R)	ESASetStrLen	(not standard ST) set S length to L chars; fill with C
HEX (V)	Hex	(not standard ST) conv. V in a string with hex format
OCT (V)	Oct	(not standard ST) conv. V in a string with oct format
BIN (V)	/	(not standard ST) conv. V in a string with bin format
ASC (C)	Asc	(not standard ST) get the ASCII code of character C
Utility		
ISSTRUCTURE (V)	/	(not standard ST) see if value V is a structure
ISFUNCTION (V)	/	(not standard ST) see if value V is a function instance
ISARRAY (V)	IsArray	(not standard ST) see if value V is an array
NUMDIM (V)	/	(not standard ST) number of dimensions of array V
LBOUND (V)	LBound	(not standard ST) lower bound of 1st dim. of array V
UBOUND (V)	UBound	(not standard ST) upper bound of 1st dim. of array V
SIZEOF (V)	/	(not standard ST) total size, in bytes, of given value
TYPEOF (V)	/	(not standard ST) name of type of given value
Conversion		
DEG_TO_RAD (V)	/	(not standard ST) convert an angle from degs to rads
RAD_TO_DEG (V)	/	(not standard ST) convert an angle from rads to degs
BCD_TO_BIN (V)	/	(not standard ST) convert an unsigned int. from BCD
BIN_TO_BCD (V)	/	(not standard ST) convert an unsigned int. to BCD
ANY_TO_SINT (V)	/	convert a value in a specific type
ANY_TO_INT (V)	CInt, CSng	convert a value in a specific type
ANY_TO_DINT (V)	CLng (see: DateToLong)	convert a value in a specific type
ANY_TO_LINT (V)	/	convert a value in a specific type
ANY_TO_USINT (V)	/	convert a value in a specific type
ANY_TO_UINT (V)	/	convert a value in a specific type
ANY_TO_UDINT (V)	/	convert a value in a specific type
ANY_TO_ULINT (V)	/	convert a value in a specific type
ANY_TO_REAL (V)	/	convert a value in a specific type
ANY_TO_LREAL (V)	CDbl	convert a value in a specific type
ANY_TO_TIME (V)	/ (similar: CDate)	convert a value in a specific type
ANY_TO_LTIME (V)	/ (similar: CDate)	convert a value in a specific type
ANY_TO_DATE (V)	/ (similar: CDate, DateValue) (see:LongToDate)	convert a value in a specific type
ANY_TO_TOD (V)	/ (similar: CDate, TimeValue)	convert a value in a specific type
ANY_TO_LTOD (V)	/ (similar: CDate, DateValue)	convert a value in a specific type
ANY_TO_DT (V)	/ (similar: CDate, DateValue)	convert a value in a specific type
ANY_TO_LDT (V)	/ (similar: CDate, DateValue)	convert a value in a specific type
ANY_TO_STRING (V)	/ (similar: CStr)	convert a value in a specific type
ANY_TO_WSTRING (V)	CStr	convert a value in a specific type

ANY_TO_CHAR (V)	/ (similar: Chr)	convert a value in a specific type
ANY_TO_WCHAR (V)	/ (similar: Chr)	convert a value in a specific type
ANY_TO_BOOL (V)	CBool	convert a value in a specific type
ANY_TO_BYTE (V)	CByte	convert a value in a specific type
ANY_TO_WORD (V)	/	convert a value in a specific type
ANY_TO_DWORD (V)	/	convert a value in a specific type
ANY_TO_LWORD (V)	/	convert a value in a specific type

SYSTEM		
Basics		
_TRACE (M)	/	print the message M on the <stdout>
_REBOOT ()	/	reboot the machine
_SHUTDOWN ()	/	shutdown the machine
RUNAPPLICATION (N,P,M,[O],[E])	/	run an application with given command line
KILLAPPLICATION (N)	/	terminate an external application of given name
RUNSCRIPT (S)	/	execute a given piece of script code
EXITRUNTIME ()	/	close the runtime application
SLEEP (T)	ESASleep	sleep for a number of ms
LSLEEP (T)	/	sleep for a number of ns
ERRORGETMESSAGE (E)	GetErrorMsg	obtain the description message of a given error
ERRORGETMODULE (E)	/	obtain the name of the module related to an error
ERRORRESET ()	/	reset the pending error code
FLUSHCONFIG ()	ESASaveStatus	flush the windows registry
FLUSHPERSISTENT ()	FlushPersistentData	flush on disk all the system persistent data
REFRESHIPADDRESSES ()	RefreshIpAddresses	refresh the IP addresses in the system variables
SETRESTAPIPREFIX ()	/	set prefix for managed HTTP API calls
SETRESTAPIRESPONSE ()	/	set a response string for HTTP API calls
SENDRESTAPIREQUEST ()	/	send an HTTP API request as client
GETRESTAPIRESPONSE ()	/	get the server response for the HTTP API sent
SETTIMESYSTEM (U)	SetTimeMode	set the time system (local/utc) used for outputs
GETTIMESYSTEM ()	/	get the time system (local/utc) used for outputs
GETNUMCLIENTSWEB ()	/	number of clients connected through web server
GETNUMCLIENTSUI ()	/	number of clients connected through web socket
GETNUMCLIENTSNET ()	/	number of clients connected in a network project
LANGUAGEGET ()	LanguageGet	retrieve the current server language
LANGUAGESET (L)	LanguageSet	set a new language in server
LANGUAGENEXT ()	LanguageNext	set the next language in server
LANGUAGEPREVIOUS ()	LanguagePrevious	set the previous language in server
GETURL (U,F,X,U,P,S,R,W)	ESAGetUrlExt	retrieve a file from an URL address
GETCLIENTID ()	/	retrieve the ID of the calling client
SETCLIENTDEBUGRIGHTS (I,R)	/	choose the debugging rights of a client
GETCLIENTDEBUGRIGHTS (I)	/	retrieve the debugging rights of a client
GETCLIENTDEBUGSTATE (I)	/	see if a client started a debug in its browser
SETCLIENTOFFSCAN (I,S)	/	set the offscan state of a client
GETCLIENTOFFSCAN (I)	/	retrieve the offscan state of a client
SETCLIENTKEYBURST (I,S)	/	set the key-burst management state of a client
GETCLIENTKEYBURST (I)	/	get the key-burst management state of a client
UI-related		
SAVESCREEN (F)	SaveScreen	save on file a screenshot of the server screen
BEEP (T,F)	ESABeep	buzzer beep of given duration and frequency
BEEPON (F)	/	switch on the buzzer
BEEPOFF ()	/	switch off the buzzer
LIGHTUP ()	/	increase the display brightness
LIGHTDOWN ()	/	decrease the display brightness
LIGHTSET (L)	/	set the display brightness to a given level
LIGHTGET ()	/	get the display brightness level
LIGHTGETMAX ()	/	get the display maximum brightness level
SHOWTASKBAR (S)	/	show and hide the taskbar
SETHHLEDSTATE (L,S)	/	switch on/off a led of a handheld panel
Clock		
GETTICKS ()	ESAClock (similar: Timer)	get the current system ticks (in ms)
GETLTICKS ()	/	get the current system ticks (in ns)

SETRTC (T) [ANY]	/ (sim: ESASetDate, --SetTime, --SetDateTime)	set a new system date and time (local mode)
SETRTC_UTC (T) [ANY]	/	set a new system date and time (UTC mode)
GETRTCTOD ()	/ (similar: Time, Now, Timer)	get the current system time (local)
GETRTCLTOD ()	/ (similar: Time, Now, Timer)	get the current system time (local)
GETRTCDATE ()	/ (similar: Date, Now)	get the current system date (local)
GETRTCDT ()	/ (similar: Date, Time, Now)	get the current system date and time (local)
GETRTCLDT ()	/ (similar: Date, Time, Now)	get the current system date and time (local)
GETRTCTOD_UTC ()	/	get the current system time (UTC)
GETRTCLTOD_UTC ()	/	get the current system time (UTC)
GETRTCDATE_UTC ()	/	get the current system date (UTC)
GETRTCDT_UTC ()	/	get the current system date and time (UTC)
GETRTCLDT_UTC ()	/	get the current system date and time (UTC)
REFRESHRTC ()	/	align runtime clock & info with system rtc
UTC_TO_LOCAL (V)	/	convert a date/time from UTC to local
LOCAL_TO_UTC (V)	/	convert a date/time from local to UTC
GETYEAR (D)	/ (similar: DatePart, Year)	extract the year from a date/time
GETMONTH (D)	/ (similar: DatePart, Month)	extract the month from a date/time
GETDAY (D)	/ (similar: DatePart, Day)	extract the day from a date/time
GETWEEKDAY (D)	/ (similar: DatePart, WeekDay)	find the weekday of a given date
GETHOURS (T)	/ (similar: DatePart, Hour)	extract the hours from a date/time
GETMINUTES (T)	/ (similar: DatePart, Minute)	extract the minutes from a date/time
GETSECONDS (T)	/ (similar: DatePart, Second)	extract the seconds from a date/time
GETMSECONDS (T)	/ (similar: DatePart)	extract the milliseconds from a date/time
GETNSECONDS (T)	/ (similar: DatePart)	extract the nanoseconds from a date/time
MAKETOD (H,M,S,m)	/ (similar: TimeSerial)	create a time with given components
MAKELTOD (H,M,S,n)	/ (similar: TimeSerial)	create a time with given components
MAKEDATE (Y,M,D)	/ (similar: DateSerial)	create a date with given components
MAKEDT (Y,M,D,H,m,S)	/ (similar: TimeSerial, DateSerial)	create a date/time with given components
MAKELDT (Y,M,D,H,m,S,n)	/ (similar: TimeSerial, DateSerial)	create a date/time with given components
Variables		
RT_PLATFORM	/	code of the machine platform architecture
RT_WORKMODE	/	code of runtime modules configuration
RT_SIMULATION	/	runtime simulation flag
RT_VERSION	/	string version of the server runtime
RT_VERSION_MAJOR	/	major component of runtime version
RT_VERSION_MINOR	/	minor component of runtime version
RT_VERSION_PROGRESS	/	progress component of runtime version
ST_VERSION	/ (similar: ScriptEngineXXXVersion, Version)	string version of the ST engine implementation
RT_SESSION	/	unique code of runtime execution session
ERRNO	/ (similar: errno, Result)	result of last executed operation
ERRMSG	/	description message of error in ERRNO
FXRESULT	LastError	last error returned by a runtime function
RESTAPIRESPONSE	/	last response received from a rest api server
UI / client		
/ (UI only)	ESAMsgBox	
/ (UI only)	ESANotifyBox	
/ (UI only)	ScreenSaverEnter	
/ (UI only)	ScreenSaverKick	

COMMON - FILES

Management		
FILE_EXIST (N)	Exists	see if a file with given name exists
FILE_COPY (S,D,O)	Copy	copy the file S in D
FILE_DELETE (N)	Delete	delete the file with name N
FILE_RENAME (S,D)	Rename	rename the file S in D
FILE_CREATEDIR (N)	MD	create a new directory N
FILE_DELETEDIR (N,R)	RD	remove the directory N; optionally recursive
Info		
FILE_GETSIZE (N)	GetFileLen	retrieve the size of the file N
FILE_SETSIZE (N,S)	SetFileLen	set (fill up or truncate) the size of the file N
FILE_GETTIMECREATION (N)	/	get the creation time of the file N
FILE_GETTIMEWRITE (N)	/	get the last write time of the file N

FILE_GETTIMEACCESS (N)	/	get the last access time of the file N
FILE_ISDIRECTORY (N)	IsDirectory	see if the given path/name identifies a directory
FILE_FINDFIRST (P)	FindFirst	start a file browse session and obtain the 1st file
FILE_FINDNEXT ()	FindNext	obtain a new file for the active browse session
FILE_FINDCLOSE ()	/	close the browse session in progress
FILE_AVAILABLESPACE (P)	AvailableSpace	retrieve the available space on a storage unit
FILE_ABSOLUTEPATH (N)	/	transform a relative or extended path
Streams		
FILE_OPEN (N,A,C)	Open	open a file with given name N, access A, mode C
FILE_CLOSE (F)	Close	close an opened file
FILE_FLUSH (F)	Commit	flush the file F buffers on disk
FILE_REWIND (F)	Rewind	reposition the file F pointer to the file start
FILE_SEEK (F,P,S)	/	set the file F pointer to a given position
FILE_ISEOF (F)	IsEOF	see if the file F reached its end
FILE_GETLENGTH (F)	GetLen	get the length of the opened file F
FILE_GETPOSITION (F)	/	get the current position of the file F pointer
FILE_WRITEENCODING (F,C)	/ (see SetUnicode)	write a unicode or UTF8 marker in a text file
FILE_READENCODING (F)	/ (see SkipUnicode)	read the encoding marker from a text file
FILE_SETENCODING (F,C)	/	force a new encoding type for a text file
FILE_GETENCODING (F)	/	retrieve the encoding type of a text file
FILE_READBYTE (F)	ReadByte	read a single byte from file F
FILE_READWORD (F)	/	read a single word (2 bytes) from file F
FILE_READDWORD (F)	/	read a double word (4 bytes) from file F
FILE_READLWORD (F)	/	read a long word (8 bytes) from file F
FILE_READBUFFER (F,L)	/	read a number of bytes from file F
FILE_READSTRING (F,L)	ReadStr	read a string of length L from file F
FILE_READLINE (F,L)	ReadLine	read a string from file F up to a line terminator
FILE_WRITE (F,V,[S])	WriteByte, WriteStr	write anything in file F (bytes, buffers, strings,…)
FILE_GETREADLENGTH (F)	/	retrieve number of bytes acquired with last read
<SEEK+READSTRING>	ReadStrIdx	
<SEEK+WRITE>	WriteStrIdx	
Variables		
FILE_NUMBER	FileCount	number of files currently opened
FILE_AUTOFLUSH	FileFlush	set to TRUE to make immediate automatic flushes
FILE_FOUNDNAME	/	returns the name of the last browsed file
FILE_FOUNDSIZE	/	returns the size of the last browsed file
FILE_FOUNDTIME	/	returns the last write time of the last browsed file
FILE_FOUNDISDIR	/	TRUE if the last browsed element is a directory

COMMON - SERIAL		
Management		
COM_OPEN (C,B,D,P,S)	Open	open a com channel over a serial port
COM_CLOSE (P)	Close	close an opened serial port
COM_FLOW (P,A,B,C,D)	/	set up a serial port flow control
COM_ISOPEN (P)	IsOpen	see if a serial port is currently open
COM_DATALENGTH (P)	IsData	see how much data is available on a serial port
COM_READBYTE (P,T)	ReadByte	read data from a serial port
COM_READBUFFER (P,S,T)	/	read data from a serial port
COM_WRITE (P,V,[S])	WriteByte, WriteStr	write data on a serial port
COM_CLEAR (P)	Clear	clear the data buffer of a serial port
COM_GETCTS (P)	GetCTS	read the CTS signal of a serial port
COM_GETDSR (P)	GetDSR	read the DSR signal of a serial port
COM_GETRING (P)	GetRing	read the Ring Indicator signal of a serial port
COM_GETRLSD (P)	GetRLSD	read the RLSD signal of a serial port
COM_SETRTS (P,S)	SetRTS, ClrRTS, Escape	change the RTS signal of a serial port
COM_SETDTR (P,S)	/ (see Escape)	change the DTR signal of a serial port
COM_SET485 (P,S)	Set485Mode	change the 485 direction of a serial port
Variables		
COM_NUMBER	/	number of serial ports currently opened
COM_RXLENGTH	/	number of bytes read from serial port
COM_TXLENGTH	/	number of bytes written on serial port

COMMON - ETHERNET

Generic		
ETH_IP (A,[B,C,D])	/	prepare an IP address
ETH_GETIP (A)	/	convert a normalized IP in different forms
ETH_PING (A,T,[N])	/	ping an IP address
TCP client		
ETH_TCPC_OPEN (A,P,[L])	Open	open a socket as TCP client
ETH_TCPC_CLOSE (E)	Close	close a socket opened as TCP client
ETH_TCPC_GETIPLOCAL (E)	GetAddress	get the local address of a TCP client
ETH_TCPC_GETIPSERVER (E)	GetServerAddress	get the address of the connected TCP server
ETH_TCPC_DATALENGTH (E)	IsData	get the amount of data available for read
ETH_TCPC_READBYTE (E,T)	/	read data from an ethernet channel
ETH_TCPC_READBUFFER (E,S,T)	ReadBuffer	read data from an ethernet channel
ETH_TCPC_READSTRING (E,S,T)	ReadString	read data from an ethernet channel
ETH_TCPC_READWSTRING (E,S,T)	/	read data from an ethernet channel
ETH_TCPC_WRITE (E,V,[S])	WriteBuffer, WriteString	write data over an ethernet channel
TCP server		
ETH_TCPS_OPEN (A,P)	Open	open a socket as TCP client
ETH_TCPS_CLOSE (E,[A,P])	Close, CloseClient	close a socket opened as TCP client
ETH_TCPS_CLIENTSNUMBER (E)	NumberOfClientsConnected	count the number of connected clients
ETH_TCPS_GETIPLOCAL (E)	GetAddress	get the local address of a TCP server
ETH_TCPS_GETIPCLIENT (E,I)	GetClientAddress	get the address of a connected TCP client
ETH_TCPS_DATALENGTH (E,[A,P])	IsData, IsDataClient	get the amount of data available for read
ETH_TCPS_READBYTE (E,T,[A,P])	/	read data from an ethernet channel
ETH_TCPS_READBUFFER (E,S,T,[A,P])	ReadBuffer, ReadBufferClient	read data from an ethernet channel
ETH_TCPS_READSTRING ('')	ReadString, ReadStringClient	read data from an ethernet channel
ETH_TCPS_READWSTRING ('')	/	read data from an ethernet channel
ETH_TCPS_WRITE (E,V,[S],A,P)	WriteBuffer, WriteString	write data over an ethernet channel
UDP		
ETH_UDP_OPEN (A,P)	Open	open a socket in UDP mode
ETH_UDP_CLOSE (E)	Close	close a socket opened in UDP
ETH_UDP_GETIPLOCAL (E)	GetAddress	get the local address of a UDP socket
ETH_UDP_DATALENGTH (E)	IsData	get the amount of data available for read
ETH_UDP_READBYTE (E,T)	/	read data from an ethernet channel
ETH_UDP_READBUFFER (E,S,T)	ReadBuffer	read data from an ethernet channel
ETH_UDP_READSTRING (E,S,T)	ReadString	read data from an ethernet channel
ETH_UDP_READWSTRING (E,S,T)	/	read data from an ethernet channel
ETH_UDP_WRITE (E,V,[S],A,P)	WriteBuffer, WriteString	write data over an ethernet channel
Variables		
ETH_ERROR	/	error code returned by ethernet functions
ETH_NUMBER	/	number of ethernet channels locally opened
ETH_RXLENGTH	/	number of bytes acquired by the last read
ETH_TXLENGTH	/	number of bytes transmitted by the last write
ETH_IPADDRESS	/	extended result for partner IP address
ETH_IPPORT	/	extended result for partner IP port
ETH_IP#	/	the 4 components of an IP address

COMMON - LIBRARIES

Standard		
LIBRARY_LOAD (N)	/ (similar: CreateObject)	load a dynamic library
LIBRARY_RELEASE (L)	/ (see CreateObject mechanics)	release a dynamic library
LIBRARY_FXLOAD (L,N,[T])	/ (see CreateObject mechanics)	declare a function of a dynamic library
LIBRARY_FXRELEASE (F)	/ (see CreateObject mechanics)	discard a function of a dynamic library
LIBRARY_FXCALL (F,...)	/ (see CreateObject mechanics)	invoke a function of a dynamic library
COM Objects		
COMLIB_LOAD (N)	CreateObject	create an instance of a COM object
COMLIB_RELEASE (I)	/ (set to Nothing)	release a COM object
COMLIB_FXLOAD (L,N,[R],[P..P])	/ (native)	declare a function of a COM interface
COMLIB_FXRELEASE (I)	/ (native)	discard a function of a COM interface
COMLIB_FXCALL (I,[P..P])	/ (native)	invoke a function of a COM object
COMLIB_PROPGET (I)	/ (native)	read the value of a COM object property
COMLIB_PROPSET (I,V)	/ (native)	write the value of a COM object property
Variants		
COMVAR_CREATE ()	/ (native)	create a 'variant' variable to use with COMs

COMVAR_DESTROY (I)	/ (native)	destroy am existing 'variant'
COMVAR_CLEANUP (I)	/ (native)	cleanup the value of a 'variant'
COMVAR_COPY (S,D)	/ (native)	copy the value from a 'variant' to another
COMVAR_DIMARRAY (I,F,T,[F,T..])	/ (native)	define a 'variant' as an array of given dimensions
COMVAR_GETNUMDIM (I)	/	get the number of dimensions of a COM array
COMVAR_GETLBOUND (I,D)	LBound	get the lower bound of a COM array dimension
COMVAR_GETUBOUND (I,D)	UBound	get the upper bound of a COM array dimension
COMVAR_SET (I,T,V)	/ (native)	write a value in a 'variant'
COMVAR_SETELEMENT (I,T,V,X,[..])	/ (native)	write a value in an element of a 'variant' array
COMVAR_GET (I,T)	/ (native)	read the value of a 'variant'
COMVAR_GETELEMENT (I,T,X,[..])	/ (native)	read the value of an element of a 'variant' array
Variables		
COMLIB_ERRNO	/	result of last invoked COM function
COMLIB_ERRMSG	/	description message of error in COMLIB_ERRNO

COMMON - PRINT

Printing		
<todo>	Start(UserFlag)	
<todo>	End()	
<todo> (print only)	Abort()	
<todo>	NewPage()	
<todo>	WriteLN(Text)	
<todo>	WriteRC(Row, Column, Text)	
<todo>	WriteXY(x, y, Text)	
<todo>	PrintImage(PathName, x, y, [w], [h])	
<todo>	SetFont(Name,Sz,[Bld],[Itl],[Uln],[Set])	
PDF		
PDF_OPEN (F)	/	start creation of a new PDF document
PDF_CLOSE ()	/	finalize creation of PDF document
PDF_NEWPAGE ()	/	finalize current page and add a new one
PDF_SETCOLOR (C)	/	set the color valid for all drawings
PDF_SETLINEWIDTH (W)	/	set the line width valid for all geometric directives
PDF_SETFONT (N,S,B,I,U)	/	set a new font with all its attributes
PDF_SETFONTNAME (N)	/	change the font face name
PDF_SETFONTSIZE (S)	/	change the font size
PDF_SETFONTBOLD (B)	/	change the font bold attribute
PDF_SETFONTITALIC (I)	/	change the font italic attribute
PDF_SETFONTUNDERLINE (U)	/	change the font underline attribute
PDF_DRAWTEXT (X,Y,T)	/	draw a text in the document
PDF_DRAWLINE (X1,Y1,X2,Y2)	/	draw a line in the document
PDF_DRAWLINEH (X,Y,L)	/	draw a horizontal line in the document
PDF_DRAWLINEV (X,Y,L)	/	draw a vertical line in the document
PDF_DRAWRECTANGLE (X,Y,X,Y,F)	/	draw a rectangle in the document
PDF_DRAWCIRCLE (X,Y,R,F)	/	draw a circle in the document
PDF_DRAWELLIPSE (X,Y,X,Y,F)	/	draw an ellipse in the document
PDF_DRAWIMAGE (X,Y,W,H,F)	/	draw an image in the document
PDF_GETTEXTWIDTH (T)	/	retrieve the width of a given text
Reports		
REPORT_EXPORT (R,N,U,G,F,T,S,L)	/	create a whole report in a PDF file
REPORT_ENABLESECTION (C,E)	/	Enable/disable classes of section of PDF reports
Variables		
PDF_PAGEWIDTH	PageWidth	width of the document pages
PDF_PAGEHEIGHT	PageHeight	height of the document pages
PDF_FONTHEIGHT	FontSize	total height of the current font
PDF_FONTASCENT	/ (see FontSize)	height of ascent part of the font
PDF_FONTDESCENT	/ (see FontSize)	height of descent part of the font
/ (doesn't apply)	PageRows	
/ (doesn't apply)	PageColumns	
/ (doesn't apply)	MarginHor	
/ (doesn't apply)	MarginVert	

COMMON - EXTERNAL

EveryWare

EW_ON ()	EverywareOn	start the everyware client
EW_OFF ()	EverywareOff	stop the everyware client
EW_ENABLE ()	EverywareEnable	enable everyware connection and functionality
EW_DISABLE ()	EverywareDisable	disable everyware connection and functionality
EW_STATUS ()	EverywareStatus	get the current everyware working state
EW_EXIST ()	EverywareExist	see if the everyware client is currently running
Messaging		
MSG_EMAIL (...)	SendMailSingle	send an e-mail to given recipients
MSG_EMAILLIST (...)	SendMailList	send an e-mail to a whole mailing list
MSG_APPNOTIFICATION (...)	/	send an app notification to given recipients
MSG_APPNOTIFICATIONLIST (...)	/	send an app notification to a whole mailing list
MSG_SMS (...)	SendSmsSingle	send an SMS to a given recipients
MSG_SMSLIST (...)	SendSmsList	send an SMS to a whole mailing list
SoftPlc		
<TBD>	CoDeSysOn	already implemented for PC, specs TBD, Linux TBC
<TBD>	CoDeSysOff	already implemented for PC, specs TBD, Linux TBC
<TBD>	CoDeSysRun	already implemented for PC, specs TBD, Linux TBC
<TBD>	CoDeSysStop	already implemented for PC, specs TBD, Linux TBC
<TBD>	CoDeSysExist	already implemented for PC, specs TBD, Linux TBC

RUNTIME - TIMERS		
Management		
TIMER_START (T)	Start	start a timer counting
TIMER_STOP (T)	Stop	stop a timer counting
TIMER_SUSPEND (T)	Suspend	suspend a timer counting
Info		
TIMER_SETLIMIT (T,L)	SetTimerValue	set a new value for the timer counter limit
TIMER_GETLIMIT (T)	GetTimerValue	obtain the current value of the counter limit
TIMER_SETPROGRESS (T,P)	SetProgress	set a new value for the timer progress counter
TIMER_GETPROGRESS (T)	GetProgress	obtain the current value of the progress counter
TIMER_ISSTARTED (T)	IsStarted	see if a timer is currently counting
TIMER_ISSUSPENDED (T)	IsSuspended	see if a timer is currently suspended

RUNTIME - TAGS		
Acquisitions		
TAG_GETVALUE (T)	GetCurrentValue, GetCurrentValue64	get the current tag value
TAG_SETVALUE (T,V)	/	change the current value of a tag
TAG_FLUSHVALUE (T)	/	send to device the current value of a tag
TAG_READVALUE (T)	ReadValue, ReadValue64	read a tag value from device
TAG_WRITEVALUE (T,V)	WriteValue, WriteValue64	write a tag value on device
TAG_READELEMENT (T,E)	ReadElement, ReadElement64	read a tag-array element from device
TAG_WRITEELEMENT (T,E,V)	WriteElement, WriteElement64	write a tag-array element on device
TAG_READBIT (T,B)	ReadBit	read a tag bit from device
TAG_WRITEBIT (T,B,V)	WriteBit	write a tag bit on device
TAG_READITEM (...)	ReadItem	read any memory item from device
TAG_WRITEITEM (...)	WriteItem	write any memory item on device
TAG_GETFIELDVALUE (T,P)	/	get the value of a field within a structured tag
TAG_SETFIELDVALUE (T,P,V)	/	change the value of a field within a structured tag
TAG_ASSIGNSTRUCT (T,S,A)	/	produce an assignable structure from a tag
TAG_FLUSH ()	/	flush on disk the persistent values of tags
Info		
TAG_GETID (N)	GetTagId	get the tag ID
TAG_GETNAME (T)	GetTagName	get the tag name
TAG_GETSHAREDID (T)	/	get the tag shared ID
TAG_GETIDFROMSHARED (T)	/	get the tag ID
TAG_GETVALUETYPE (T)	GetTagValueType	get the code of the tag value type
TAG_GETSTRLENGTH (T)	GetTagStrLength	get the string length, in case of string value types
TAG_GETARRAYSIZE (T)	GetTagArraySize	get the number of elements of a tag array
TAG_GETDEVICEID (T)	GetTagDeviceId	get the ID of the tag's device
TAG_GETAREAID (T)	GetTagAreaId	get the ID of the tag's data area
TAG_GETADDRESS (T,I)	GetTagAddress	get an address field of the tag
TAG_GETFIELDOFFSET (T,P)	GetFieldOffset	get the offset of a field within a structured tag
TAG_GETFIELDADDRESS (T,P)	GetFieldAddress	get the address of a field of a structured tag

TAG_GETNUMBERALL ()	/	count the total number of tags
TAG_GETNUMBEREXT ()	/	count the number of external tags
TAG_GETCLIENTTAGNAME (I)	/	get the name of a client system tag
TAG_ISOFFLINE (T)	IsOffline	get the offline state of a tag
TAG_ISOFFSCAN (T)	/	get the offscan state of a tag
TAG_SETOFFSCAN (T,S)	SetTagOffscan	set the offscan state of a tag
TAG_SETOFFSCANDEV (D,S)	SetDeviceOffscan	set the offscan state of a device
TAG_DEVICESNUMBER ()	/	get the number of configured devices
TAG_DEVICEGETID (N)	GetDeviceId	get the ID of a device
TAG_DEVICEGETNAME (D)	GetDeviceName	get the name of a device
TAG_AREASNUMBER (D)	/	get the number of a device's data areas
TAG_AREAGETID (D,N)	/	get the ID of a device's data area
TAG_AREAGETNAME (D,A)	/	get the name of a device's data area

RUNTIME - ALARMS		
Management		
ALARM_ON (A,[U,S])	AlarmOn	raise an alarm
ALARM_OFF (A,[U,S])	ClearAlarm	clear an alarm
ALARM_ACKSINGLE (I,[U,S])	AckAlarm	acknowledge a single alarm instance
ALARM_ACKINSTANCES (A,[U,S])	AckInstances	acknowledge all the instances of a given alarm
ALARM_ACKGROUP (G,[U,S])	AckGroup	acknowledge all the alarms of a given group
ALARM_ACKALL ([U,S])	AckGlobal	acknowledge all the existing alarm instances
ALARM_ISON (A)	IsAlarmOn	see if an alarm condition is currently set
ALARM_HISTORYRESET ()	HistoryDelete	reset the history records
ALARM_HISTORYFLUSH ()	HistoryFlush	flush on persistent storage the alarms history
ALARM_STATSRESET ()	StatsDelete	reset the alarms statistical information
ALARM_STATSFLUSH ()	StatsFlush	flush on persistent storage the alarms statistics
ALARM_EXPORT (F)	AlarmsExport	export on file the active alarms
ALARM_EXPORTHISTORY (F)	HistoryExport	export on file the historical records
ALARM_EXPORTSTATS (F)	StatsExport	export on file the statistical information
ALARM_EXPORTCONFIG (A,H,S)	/	change the export fields keys of alarms
ALARM_PRINT ()	AlarmsPrint	print the active alarms
ALARM_PRINTHISTORY ()	HistoryPrint	print the historical records
ALARM_PRINTSTATS ()	StatsPrint	print the statistical information
Info		
ALARM_GETNUMBER ([P])	/	get the number of active instances
ALARM_GETNUMISA ([P])	/	get the number of active ISA instances
ALARM_GETNUMEVENTS ([P])	/	get the number of active simple events
ALARM_GETNUMACK ([P])	/	get the number of ISA still waiting for ACK
ALARM_GETNUMHISTORY ()	/	get the number of records in history
ALARM_GETNUMINSTANCES (A)	CountAlarmInstances	get the number of instances of a given alarm
ALARM_GETINSTANCEID (A,I)	GetInstanceId	get the ID of a given alarm instance
ALARM_GETINFO ([A],[I])	/	get information parameters of a given alarm
ALARM_GETNAMEFROMKEY (K)	/	get the name of an alarm with given key
ALARM_GETIDFROMKEY (K)	/	get the ID of an alarm with given key
ALARM_GETMSGFROMKEY (K)	/	get the message of an alarm with given key
ALARM_GETFIRSTPRJ ()	/	get info about the first project alarm and stats
ALARM_GETNEXTPRJ (I)	/	get info about the project alarms and stats
ALARM_GETFIRSTON ()	/	get info about the first project alarm in on state
ALARM_GETNEXTON (I)	/	get info about the project alarms in on state
ALARM_GETFIRSTACTIVE ()	/	get info about the first active alarm
ALARM_GETNEXTACTIVE (I)	/	get info about the active alarms
ALARM_GETFIRSTHISTORY ()	/	get info about the first history record
ALARM_GETNEXTHISTORY (I)	/	get info about the history records
ALARM_GETFIRSTHPACK ()	/	get info about the first packed history record
ALARM_GETNEXTHPACK (I)	/	get info about the packed history records
Variables		
ALARM_NAME	/	name of the analyzed alarm
ALARM_ID	/	ID of the analyzed alarm
ALARM_KEY	/	custom key of the analyzed alarm
ALARM_MESSAGE	/	description message of the analyzed alarm
ALARM_RECNAME	/	information filled by alarms browse loops

ALARM_RECKEY	/	information filled by alarms browse loops
ALARM_RECID	/	information filled by alarms browse loops
ALARM_RECSTATE	/	information filled by alarms browse loops
ALARM_RECNUMINSTANCES	/	information filled by alarms browse loops
ALARM_RECONNUMBER	/	information filled by alarms browse loops
ALARM_RECONDURATION	/	information filled by alarms browse loops
ALARM_RECINSTANCEID	/	information filled by alarms browse loops
ALARM_RECMESSAGE	/	information filled by alarms browse loops
ALARM_RECONTIME	/	information filled by alarms browse loops
ALARM_RECONUSER	/	information filled by alarms browse loops
ALARM_RECALTTIME	/	information filled by alarms browse loops
ALARM_RECALTUSER	/	information filled by alarms browse loops
ALARM_RECEVENTTYPE	/	information filled by alarms browse loops
ALARM_RECEVENTTIME	/	information filled by alarms browse loops
ALARM_RECEVENTUSER	/	information filled by alarms browse loops
UI / client		
/ (client only)	AlarmsExportLocal	
/ (client only)	HistoryExportLocal	
/ (client only)	StatsExportLocal	
/ (client only)	AlarmsPrintLocal	
/ (client only)	HistoryPrintLocal	
/ (client only)	StatsPrintLocal	

RUNTIME - RECIPES		
Transfers		
RECIPE_LOAD (S,R,[L])	LoadRecipe	load a recipe from archive to buffer
RECIPE_SAVE (S,[R],[L])	SaveRecipe, SaveRecipeAs	save a recipe from buffer to archive
RECIPE_DOWNLOAD (S,R,[Y],[L])	RecipeDownload	transfer a recipe from archive to device
RECIPE_UPLOAD (S,[R],[Y],[L])	RecipeUpload	transfer a recipe from device to archive
RECIPE_DOWNLOADBUF (S,[Y],[L])	RecipeBufferDownload	transfer a recipe from buffer to device
RECIPE_UPLOADBUF (S,[Y],[L])	RecipeBufferUpload	transfer a recipe from device to buffer
RECIPE_TRANSFERBUSY ()	Busy (variable)	see if a recipes transfer is in progress
RECIPE_TRANSFERWAIT ()	/	wait for termination of a transfer in progress
Management		
RECIPE_DELETE (S,[R],[L])	DeleteRecipe, DeleteAllRecipes	delete one or all the recipes of a structure
RECIPE_RENAME (S,O,N,[L])	RenameRecipe	rename a given recipe
RECIPE_PACKARCHIVE (S)	PackArchive	compact the archive of a given structure
RECIPE_CLEARBUFFER (S)	ClearTagBuffer	clear the buffer tags of a given structure
RECIPE_COMPARE (S,A,B)	RecipeCompare	compare two recipes of a given structure
RECIPE_COMPARESET (S,N,D,R,[T])	/	compare a recipe in archive with a tags set
RECIPE_COMPAREFIELD (S,R,F,[M])	/	compare a field in archive with its device tag
RECIPE_EXPORT (F,[S],[L])	RecipeExport, RecipeExportAll	export the recipes of one or all the structures
RECIPE_EXPORTFLAT (F,S,[L])	RecipeExportCsv	export the recipes of one or all the structures
RECIPE_IMPORT (F,[S],[L])	RecipeImport, RecipeImportAll	import the recipes of one or all the structures
RECIPE_PRINT ([S])	RecipePrint, RecipePrintAll	print the recipes of one or all the structures
Info		
RECIPE_GETCURNAME (S)	/	get the name currently in the buffer of a structure
RECIPE_EXIST (S,R)	RecipeExists	see if a given recipe exists in the archive
RECIPE_GETNUMBER (S)	GetRecipeCount	count the valid recipes in a structure archive
RECIPE_GETRECORDS (S)	GetRecipeRecords	count all the records in a structure archive
RECIPE_GETINFO (S,I)	GetRecipeName	get the name of a recipe of known ID
RECIPE_GETID (S,R)	/	get the ID of a recipe of known name
RECIPE_GETFIELDSNUMBER (S)	/	get the number of fields in structure
RECIPE_GETFIELDNAME (S,F,[M])	/	get the name of a recipe field
RECIPE_GETFIELDINDEX (S,F)	/	get the index of a recipe field
RECIPE_GETCOMPAREINDEX (...)	/	get the index of a field comparison flag
RECIPE_GETFIELDVALUE (S,R,F)	/	get the value of a recipe field from archive
RECIPE_SETFIELDVALUE (S,R,F,V)	/	set the value of a recipe field in archive
RECIPE_GETSTRRECORD (S,R,T)	/	get a whole recipe record from archive
RECIPE_GETSTRRECORDS (S,T,Z,P[...])	/	get a set of recipe records from archive
RECIPE_GETTAGNAME (S,F,D)	GetTagName	get the name of a tag associated to a recipe field
Variables		
RECIPE_IMPORTEDNEW	ImportedNew	count the recipes added by a file import

RECIPE_IMPORTEDOLD	ImportedOld	count the recipes replaced by a file import
RECIPE_NAME	/	replicates the name of the retrieved recipe
RECIPE_ID	/	replicates the ID of the retrieved recipe
RECIPE_COMMENT	/	comment associated to the retrieved recipe
RECIPE_TIME	/	last save time of the retrieved recipe
UI / client		
/ (UI only)	RecipeLoadBox	
/ (UI only)	RecipeSaveBox	
/ (UI only)	RecipeSaveAsBox	
/ (UI only)	RecipeDeleteBox	
/ (UI only)	RecipeRenameBox	
/ (UI only)	RecipeDownloadBox	
/ (UI only)	RecipeExport (box)	
/ (UI only)	RecipeImport (box)	
/ (client only)	RecipeExportLocal	
/ (client only)	RecipeExportAllLocal	
/ (client only)	RecipePrintLocal	
/ (client only)	RecipePrintAllLocal	

RUNTIME - SAMPLES		
Management		
DLOG_ENABLE (B)	Enable	enable activity of a datalog buffer
DLOG_DISABLE (B)	Disable	disable activity of a datalog buffer
DLOG_RESETSAMPLES (B)	ResetSamples	clean up the content of a datalog buffer
DLOG_ACQUIRESAMPLES (B)	AcquireSample	acquire a set of samples for a datalog buffer
DLOG_ACQUISITIONBUSY (B)	/	see if there is an acquisition in progress
DLOG_ACQUISITIONWAIT (B)	/	wait for an acquisition to complete
DLOG_APPENDSAMPLES (B,...)	/	add a new set of samples to the buffer
DLOG_FLUSH ([B])	FlushPersistentData	flush a samples buffer on persistent storage
DLOG_EXPORT (F,B)	ExportSamples	export the samples of a datalog buffer
DLOG_PRINT (B)	SamplesPrint	print the samples of a datalog buffer
DLOG_EXPORTBUSY ([B])	ExportInProgress	see if there is an export in progress
DLOG_EXPORTTERMINATE ([B])	TerminateExport	terminate the export of a datalog buffer
DLOG_EXPORTWAIT ([B])	WaitForExport	wait for a datalog buffer export termination
DLOG_EXPORTCONFIG (B,K)	/	change the export fields key of a datalog buffer
Info		
DLOG_ISENABLED (B)	/	see if a buffer activity is currently enabled
DLOG_GETNUMSAMPLES (B)	/	get the number of samples stored in a buffer
DLOG_GETSAMPLE (B,S,I)	/	retrieve the value of a sample
DLOG_GETDISCARDINVALID (B)	/	get the current invalid samples management
DLOG_SETDISCARDINVALID (B,S)	/	set a behaviour for invalid samples management
Variables		
DLOG_SAMPLEVALUENUM	/	get the value of the last sample query
DLOG_SAMPLEVALUESTR	/	get the value of the last sample query
DLOG_SAMPLEISSTRING	/	see if the value of the last sample is a string
DLOG_SAMPLETIME	/	get the timestamp of the last sample query
DLOG_SAMPLEQUALITY	/	get the quality of the last sample query
UI / client		
/ (client only)	ExportSamplesLocal	
/ (client only)	ExportInProgressLocal	
/ (client only)	WaitForExportLocal	
/ (client only)	TerminateExportLocal	
/ (client only)	ImportSamplesLocal	
/ (client only)	SamplesPrintLocal	

RUNTIME - USERS		
Management		
USER_ADD (U,G,P,M,[V],[L],[E],[T],[R])	Add	add a new user with given properties
USER_REMOVE (U)	Remove	remove an existing user
USER_SETPASSWORD (U,P,M)	ChangePassword	change the password of a user
USER_SETVALIDITY (U,V)	ChangePasswordValidity	change the validity of a password
USER_SETGROUP (U,G)	ChangeGroup	change the group of a user
USER_SETSIGNATURE (U,S)	/	change the electronic signature string of a user

USER_SETRFID (U,R)	/	change the RFID code string of a user
USER_SETLANGUAGE (U,L)	ChangeLanguage	change the default language of a user
USER_SETEMAIL (U,E)	ChangeEmail	change the e-mail address of a user
USER_SETTELNUMBER (U,T)	ChangeTelNumber	change the telephone number of a user
USER_LOCK (U)	UserLock	lock a user
USER_UNLOCK (U)	UserUnlock	unlock a user
USER_PERMANENTLOCK (U)	/	permanently lock a user
USER_JOINLIST (U,L,T)	UserJoinList	add a user to a mailing list
USER_LEAVELIST (U,L)	UserLeaveList	remove a user from a mailing list
USER_RESETLISTS (U)	UserResetLists	remove a user from all of its mailing lists
Info		
USER_GETCURRENTNAME ()	GetCurrentUserName	get the name of the current server user
USER_GETCURRENTGROUP ()	GetCurrentGroup	get the group of the current server user
USER_GETCURRENTSHOW ()	GetCurrentVisibility	get the visibility level of the server user
USER_GETCURRENTUSE ()	GetCurrentInteractivity	get the interactivity level of the server user
USER_GETGROUP (U)	GetUserGroup	get the group of a user
USER_GETLEVELSHOW (U)	GetUserVisibility	get the visibility level of a user
USER_GETLEVELUSE (U)	GetUserInteractivity	get the interactivity level of a user
USER_GETLANGUAGE (U)	GetUserLanguage	get the default language of a user
USER_GETEMAIL (U)	GetUserEmail	get the e-mail address of a user
USER_GETTELNUMBER (U)	GetUserTelNumber	get the telephone number of a user
USER_GETVALIDITY (U)	GetUserPasswordValidity	get the validity time of a user password
USER_GETCREATION (U)	/	get the creation date of a user password
USER_GETEXPIRATION (U)	/	get the expiration date of a user password
USER_ISLOCKED (U)	/	see if a user is locked
USER_ISIMPORTED (U)	/	see if a user was imported from an AD
USER_HASRFID (U)	/	see if a user supports RFID authentication
USER_GROUPGETNAME (I)	/	get the name of a users' group
USER_GROUPGETID (G)	/	get the ID of a users' group
USER_GROUPLEVELSHOW (G)	/	get the visibility level of a users' group
USER_GROUPLEVELUSE (G)	/	get the interactivity level of a users' group
Utilities		
USER_FLUSH ()	UsersFlush	flush the log of the users' events
USER_EXPORT (N,M,[F],[T])	LogExport	export the log of the users' events
USER_PRINT ([F],[T])	UsersPrint	print the log of the users' events
USER_RESET ()	/	reset users and passwords to project defaults
USER_IMPORTNETWORK (U,W,...)	ImportNetworkUsers	import users from an Active Directory
USER_EXPORTGROUPMATRIX ()	/	export group permissions matrix
USER_IMPORTGROUPMATRIX ()	/	import group permissions matrix
USER_EXPORTGEOMATRIX ()	/	export geographic permissions matrix
USER_IMPORTGEOMATRIX ()	/	import geographic permissions matrix
UI / client		
/ (client only)	Login	
/ (client only)	Logout	
/ (UI only)	LoginBox	
/ (UI only)	LoginPasswordBox	
/ (UI only)	AddBox	
/ (UI only)	RemoveBox	
/ (UI only)	ChangeInfoBox	
/ (UI only)	SendMailBox	
/ (UI only)	SendSmsBox	
/ (client only)	LogExportLocal	
/ (client only)	UsersPrintLocal	

RUNTIME - PIPELINES		
Management		
PIPELINE_ENABLE (P)	Enable	enable the auto activity of the pipeline
PIPELINE_DISABLE (P)	Disable	disable the auto activity of the pipeline
PIPELINE_WRITE (P)	Execute	force the execution of the pipeline read/write
Info		
PIPELINE_ISENABLED (P)	IsEnabled	see if the pipeline automatic activity is enabled
PIPELINE_GETID (N)	/	get the ID of the pipeline with given name
PIPELINE_GETNAME (P)	/	get the name of the pipeline with given ID

PIPELINE_GETNUMBER ()	/	get the number of existing pipelines

RUNTIME - FDA AUDITOR		
Management		
AUDIT_ENABLE ()	EventsTracingEnable	enable the FDA auditor logging
AUDIT_DISABLE ()	EventsTracingDisable	disable the FDA auditor logging
AUDIT_FLUSH ()	EventsTracingFlush	flush on disk the records logged by FDA auditor
AUDIT_EXPORT (N,M,S,[F],[T])	EventsTracingExport, /Part	export (all or part of) the logged records
AUDIT_RESET (N,M,S)	EventsTracingReset	export and reset the logged records and errors
AUDIT_PRINT ([F],[T])	EventsPrint	print (all or part of) the logged records
Info		
AUDIT_ISENABLED ()	/	see if the FDA auditor logging is enabled
AUDIT_GETERROR ()	/	retrieve the code of an auditor blocking error
AUDIT_GETNUMBER ()	/	get the number of the records in the log
AUDIT_READ (I)	/	read a record from the FDA auditor logs
Variables		
AUDIT_READMODULE	/	module related to the read record
AUDIT_READACTION	/	action related to the read record
AUDIT_READTIME	/	logging time of the read record
AUDIT_READUSER	/	name of user related to the read record
AUDIT_READCLIENT	/	name of client related to the read record
AUDIT_READOBJECT	/	runtime object related to the read record
AUDIT_READCOMMENT	/	comment appended to the read record
AUDIT_READREASON	/	reason entered by user at runtime
AUDIT_READSIGNATURE	/	electronic signature entered by user
AUDIT_READTAGNAME	/	name of involved tag (for tag events)
AUDIT_READTAGADD	/	symbolic address of involved tag (for tag events)
AUDIT_READOLDVALUE	/	old tag value (for tag editing)
AUDIT_READNEWVALUE	/	new tag value (for tag editing)
UI / client		
/ (client only)	EventsTracingExportLocal	
/ (client only)	EventsTracingExportPartLocal	
/ (client only)	EventsTracingResetLocal	
/ (client only)	EventsPrintLocal	

UI MODULES		
Page manager		
/ (UI only)	ShowPage	
/ (UI only)	SetPageColor	
/ (UI only)	GetPageColor	
/ (UI only)	GetPageWidth	
/ (UI only)	GetPageHeight	
/ (UI only)	ShowPageNext	
/ (UI only)	ShowPageNextFull	
/ (UI only)	ShowPageNextPopup	
/ (UI only)	ShowPagePrevious	
/ (UI only)	ShowPagePreviousFull	
/ (UI only)	ShowPagePreviousPopup	
/ (UI only)	ShowPageLast	
/ (UI only)	ClosePopUp	
/ (UI only)	ClosePopUpTop	
/ (UI only)	ClosePopUpAll	
/ (UI only)	ShowHelpPage	
/ (UI only)	ShowHelpFullscreen	
/ (UI only)	ShowHelpPopup	
/ (UI only)	CloseHelpPage	
/ (UI only)	CloseHelpPages	
/ (UI only)	DisableInteraction	
/ (UI only)	EnableInteraction	
/ (UI only)	IsPageOpen	
/ (UI only)	GetPageName	
/ (UI only)	GetPageId	
/ (UI only)	GetFullScreenName	

/ (UI only)	GetFullScreenId	
/ (UI only)	GetNumPopups	
/ (UI only)	GetPopupName	
/ (UI only)	GetPopupId	
/ (UI only)	ShowRoadMap	
/ (UI only)	ShowPopupMap	
/ (UI only)	ShowSequenceRoll	
/ (UI only)	ShowDateTimeBox	
/ (UI only)	ShowResourceMonitorBox	
/ (UI only)	ShowCalculatorBox	
/ (UI only)	MeasuresNext	
/ (UI only)	MeasuresPrevious	
/ (UI only)	MeasuresSet	
/ (UI only)	MeasuresGet	
Controls		
/ (UI only)	SetRangeColor	
/ (UI only)	GetRangeColor	
/ (UI only)	SetRangeValue	
/ (UI only)	GetRangeValue	
/ (UI only)	SetMoveState	
/ (UI only)	GetMoveStateLeft	
/ (UI only)	GetMoveStateTop	
/ (UI only)	GetMoveStateAngle	
/ (UI only)	GetMoveStateTime	
/ (UI only)	SetImage	
/ (UI only)	GetImage	
/ (UI only)	SetText	
/ (UI only)	SetTextKey	
/ (UI only)	GetText	
/ (UI only)	SetPointCoord	
/ (UI only)	SetTrendTraceMinX	
/ (UI only)	SetTrendTraceMaxX	
/ (UI only)	SetTrendTraceMinY	
/ (UI only)	SetTrendTraceMaxY	
/ (UI only)	GetTrendMinX	
/ (UI only)	GetTrendMaxX	
/ (UI only)	GetTrendMinY	
/ (UI only)	GetTrendMaxY	
/ (UI only)	TraceImportLocal	
Variables		
/ (UI only)	< all control-related properties >	
/ (UI only)	< all UI-related variables >	

3. STANDARD ST LANGUAGE

Mathematical functions

ADD

Calculates the sum of all the given parameters.

RESULT = **ADD** (v1, v2 [, … , vN])

Same as:
`V1 + V2 + … + VN`

input

v# :	ANY_PLAIN	terms of the addition; several different types are allowed: the limitations are the same as the "+" operator (see [1] for details); similarly to the "+" operator, also, adding strings will result in their concatenation

output

RESULT :	ANY_PLAIN	the type of the result depends on the types of the involved operands; as above, the same rules of the "+" operator apply (see [1] for details)

SUB

Calculates the difference between v1 and v2.

RESULT = **SUB** (v1, v2)

Same as:
`V1 - V2`

input

v1,v2 :	ANY_PLAIN	terms of the subtraction; several different types are allowed: the limitations are the same as the "-" operator (see [1] for details);

output

RESULT :	ANY_PLAIN	the type of the result depends on the types of the involved operands; as above, the same rules of the "-" operator apply (see [1] for details)

MUL

Calculates the multiplication result of all the given parameters.

RESULT = **MUL** (v1, v2 [, … , vN])

Same as:
`V1 * V2 * … * VN`

input		
v# :	ANY_PLAIN	factors of the multiplication; several different types are allowed: the limitations are the same as the "*" operator (see [1] for details)

output		
RESULT :	ANY_PLAIN	the type of the result depends on the types of the involved operands; as above, the same rules of the "*" operator apply (see [1] for details)

DIV

Calculates the division between v1 and v2.

RESULT = **DIV** (v1, v2)

Same as:
`V1 / V2`

input		
v1,v2 :	ANY_PLAIN	dividend and divisor; several different types are allowed: the limitations are the same as the "/" operator (see [1] for details);

output		
RESULT :	ANY_PLAIN	the type of the result depends on the types of the involved operands; as above, the same rules of the "/" operator apply (see [1] for details)

MOD

Calculates the remainder of the division between v1 and v2.

RESULT = **MOD** (v1, v2)

Same as:
`V1 % V2` (or `V1 MOD V2`)

input		
v1,v2 :	ANY_INT	dividend and divisor; several different types are allowed: the limitations are the same as the "MOD" operator (see [1] for details);

output		
RESULT :	ANY_INT	the type of the result depends on the types of the involved operands; as above, the same rules of the "MOD" operator apply (see [1] for details)

ABS

Calculates the absolute value of the given parameter.

RESULT = **ABS** (v)

input

V :	ANY_NUM	source value

output

RESULT :	ANY_NUM	absolute value of V : |V| in case of signed integers, the result is a higher order signed integer

SQRT

Calculates the square root of the given parameter.

RESULT = **SQRT** (RADICAND)

input

RADICAND :	ANY_NUM	the root radicand; function domain : RADICAND ≥ 0

output

RESULT :	LREAL	square root of RADICAND : $\sqrt{\text{RADICAND}}$

LN

Calculates the natural logarithm of the given parameter.

RESULT = **LN** (V)

input

V :	ANY_NUM	source value; function domain : V > 0

output

RESULT :	LREAL	natural logarithm (base *e*) of V : $\log_e(V)$

LOG

Calculates the base 10 logarithm of the given parameter.

RESULT = **LOG** (V)

input

V :	ANY_NUM	source value; function domain : V > 0

output

RESULT :	LREAL	base-10 logarithm of V : $\log_{10}(V)$

EXP

Calculates the value of *e* raised to a given power (where '*e*' is the Euler's number).

RESULT = **EXP** (EXPONENT)

Same as:
`e ** EXPONENT`

input

EXPONENT :	ANY_NUM	exponent of the exponentiation

output

RESULT :	LREAL	exponentiation result : $e^{EXPONENT}$

POW

Calculates the value of a given base raised to a given power.

RESULT = **POW** (BASE, EXPONENT)

Same as:
`BASE ** EXPONENT`

input

BASE :	ANY_NUM	base of the exponentiation
EXPONENT :	ANY_NUM	exponent of the exponentiation

output

RESULT :	LREAL	exponentiation result : $BASE^{EXPONENT}$

SIN

Calculates the sine of a given angle.

RESULT = **SIN** (ANGLE)

input

ANGLE :	ANY_NUM	the angle, in radians

output

RESULT :	LREAL	sine result

COS

Calculates the cosine of a given angle.

RESULT = **COS** (ANGLE)

input

ANGLE :	ANY_NUM	the angle, in radians

output

RESULT :	LREAL	cosine result

TAN

Calculates the tangent of a given angle.

RESULT = **TAN** (ANGLE)

input		
ANGLE :	ANY_NUM	the angle, in radians

output		
RESULT :	LREAL	tangent result

ASIN

Calculates the arc-sine of a given value.

RESULT = **ASIN** (V)

input		
V :	ANY_NUM	source value; ideally a 'sine' value; function domain : -1 ≤ V ≤ 1

output		
RESULT :	LREAL	arc-sine result

ACOS

Calculates the arc-cosine of a given value.

RESULT = **ACOS** (V)

input		
V :	ANY_NUM	source value; ideally a 'cosine' value; function domain : -1 ≤ V ≤ 1

output		
RESULT :	LREAL	arc-cosine result

ATAN

Calculates the arc-tangent of a given value.

RESULT = **ATAN** (V)

input		
V :	ANY_NUM	source value; ideally a 'tangent' value

output		
RESULT :	LREAL	arc-tangent result

FRACTION

Retrieves the fractional part of a given (floating-point) value.

RESULT = **FRACTION** (V)

Similar to (range aside):
`V - ANY_TO_LINT(V)`

input		
V :	ANY_REAL	source value

output		
RESULT :	ANY_REAL	fractional part of V; replicates the type of the parameter

TRUNC

Converts a floating-point value in an integer; decimals are truncated.

RESULT = **TRUNC** (V)

Works as a plain **ANY_TO_LINT** (in case of floating-point parameter).

input		
V :	ANY_REAL	source value; the range is expected to fit in the result LINT type

output		
RESULT :	LINT	the truncated value

ROUND

Converts a floating-point value in an integer; decimals are rounded.

RESULT = **ROUND** (V)

input		
V :	ANY_REAL	source value; the range is expected to fit in the result LINT type

output		
RESULT :	LINT	the rounded value

MIN

Retrieves the minimum among all the given parameters.

RESULT = **MIN** (V1, V2 [, … , VN])

input		
V# :	ANY_PLAIN	source values to be checked; several different types are allowed and can be mixed up: the limitations are the same as the "<=" operator (see [1] for details)

output		
RESULT :	ANY_PLAIN	the minimum among all the submitted values;

returns the identified minimum with its exact value and type

MAX

Retrieves the maximum among all the given parameters.

RESULT = **MAX** (V1, V2 [, ... , VN])

input		
V# :	ANY_PLAIN	source values to be checked; several different types are allowed and can be mixed up: the limitations are the same as the ">=" operator (see [1] for details)

output		
RESULT :	ANY_PLAIN	the maximum among all the submitted values; returns the identified maximum with its exact value and type

LIMIT

Limits a given value between a minimum and a maximum.

RESULT = **LIMIT** (MINIMUM, VALUE, MAXIMUM)

Similar to the complex C combination:

```
((VALUE < MINIMUM) ? MINIMUM : ((VALUE > MAXIMUM) ? MAXIMUM : VALUE))
```

input		
MINIMUM :	ANY_PLAIN	the minimum allowed value
VALUE :	ANY_PLAIN	the value to be limited
MAXIMUM :	ANY_PLAIN	the maximum allowed value several different types are allowed and can be mixed up: for the relations between the three parameters, the limitations are the same as the "<" and ">" operators (see [1] for details)

output		
RESULT :	ANY_PLAIN	the limited value (either a copy of MINIMUM, MAXIMUM or VALUE); the returned type as well replicates that of the source parameter

SEL

Returns one of two parameters, according to the value of a given condition.

RESULT = **SEL** (CONDITION, IFFALSE, IFTRUE)

Similar to the C ternary operator:

```
((CONDITION) ? IFTRUE : IFFALSE)
```

input		
CONDITION :	BOOL	a boolean condition used to select among the other two parameters
IFFALSE :	ANY	the value returned if the CONDITION value is FALSE
IFTRUE :	ANY	the value returned if the CONDITION value is TRUE

output

RESULT :	ANY	the value selected by the condition (either a copy of IFFALSE or IFTRUE); the returned type as well replicates that of the source parameter

MUX

Returns one of the given parameters, according to the value of a selector.

RESULT = **MUX** (SELECTOR, V1, V2 [, … , VN])

Similar to the complex C combination:
`((SELECTOR == 1) ? V1 : ((SELECTOR == 2) ? V2 : ((SELECTOR == 3) ? V3 : (…))))`
In other words, the function returns the "SELECTOR-th" (+1) parameter value.

input

SELECTOR :	ANY_INT	an integer value used to select one of the other parameters; used as a base-1 index of the selectable parameters; must be in the range : 1 ≤ SELECTOR ≤ N where N is the number of V# parameters
V# :	ANY	list of selectable values

output

RESULT :	ANY	the value selected by the SELECTOR (a copy of one of the V# parameters); the returned type as well replicates that of the source parameter

RND

Obtains a pseudo-random value.

RESULT = **RND** ()

output

RESULT :	LREAL	the generated pseudo-random value; will be in the range : 0 ≤ RESULT ≤ 1

Bits functions

SHL

Shifts a given (bitstring) value to the left of a number of bits.

RESULT = **SHL** (V, BITS)

Example:
```
SHL(ANY_TO_BYTE(2#11001100)) = 2#10011000
```

input		
V :	ANY_BIT	the bitstring value to be shifted; "to the left" means toward higher bits; the declared family type is ANY_BIT but BOOLs are actually not allowed
BITS :	ANY_INT	the number of bits of the shift

output		
RESULT :	ANY_BIT	the shifted bitstring value; replicates the type of the input value; overflowed bits are lost

SHR

Shifts a given (bitstring) value to the right of a number of bits.

RESULT = **SHR** (V, BITS)

Example:
```
SHR(ANY_TO_BYTE(2#10011001)) = 2#1001100
```

input		
V :	ANY_BIT	the bitstring value to be shifted; "to the right" means toward lower bits; the declared family type is ANY_BIT but BOOLs are actually not allowed
BITS :	ANY_INT	the number of bits of the shift

output		
RESULT :	ANY_BIT	the shifted bitstring value; replicates the type of the input value; overflowed bits are lost

ROL

Shifts (rotates) a given (bitstring) value to the left of a number of bits.

RESULT = **ROL** (V, BITS)

Example:
```
ROL(ANY_TO_BYTE(2#11001100)) = 2#10011001
```

input

V :	ANY_BIT	the bitstring value to be rotated; "to the left" means toward higher bits; the declared family type is ANY_BIT but BOOLs are actually not allowed
BITS :	ANY_INT	the number of bits of the shift

output

RESULT :	ANY_BIT	the rotated bitstring value; replicates the type of the input value; overflowed bits are rolled around and placed in the lowest bits of the result

ROR

Shifts (rotates) a given (bitstring) value to the right of a number of bits.

RESULT = **ROR** (V, BITS)

Example:
```
ROR(ANY_TO_BYTE(2#10011001)) = 2#11001100
```

input

V :	ANY_BIT	the bitstring value to be rotated; "to the right" means toward lower bits; the declared family type is ANY_BIT but BOOLs are actually not allowed
BITS :	ANY_INT	the number of bits of the shift

output

RESULT :	ANY_BIT	the rotated bitstring value; replicates the type of the input value; overflowed bits are rolled around and placed in the highest bits of the result

AND

Applies a bitwise AND to all the given operands.

RESULT = **AND** (V1, V2 [, … , VN])

Same as:
`V1 & V2 & … & VN` (or `V1 AND V2 AND … AND VN`)

input

V# :	ANY_PLAIN	the values to be combined in AND; the limitations are the same as the "AND" (&) operator (see [1] for details) ⇒ - only bitstrings and unsigned integers are allowed to participate in the operation

output

RESULT :	ANY_PLAIN	the result of the operation; the type of the result depends on the types of the involved operands; as above, the same rules of the "AND" (&) operator apply (see [1] for details) ⇒ - if only bitstrings are used, the result is a bitstring of the larger involved size; - if also unsigned integers are used, the result is an unsigned integer of the larger involved size

OR

Applies a bitwise OR to all the given operands.

RESULT = **OR** (v1, v2 [, … , vN])

Same as:
`V1 | V2 | … | VN` (or `V1 OR V2 OR … OR VN`)

input		
v# :	ANY_PLAIN	the values to be combined in OR; the limitations are the same as the "OR" (|) operator (see [1] for details) ⇒ - only bitstrings and unsigned integers are allowed to participate in the operation

output		
RESULT :	ANY_PLAIN	the result of the operation; the type of the result depends on the types of the involved operands; as above, the same rules of the "OR" (|) operator apply (see [1] for details) ⇒ - if only bitstrings are used, the result is a bitstring of the larger involved size; - if also unsigned integers are used, the result is an unsigned integer of the larger involved size

XOR

Applies a bitwise XOR to all the given operands.

RESULT = **XOR** (v1, v2 [, … , vN])

Same as:
`V1 ^ V2 ^ … ^ VN` (or `V1 XOR V2 XOR … XOR VN`)

input		
v# :	ANY_PLAIN	the values to be combined in XOR; the limitations are the same as the "XOR" (^) operator (see [1] for details) ⇒ - only bitstrings and unsigned integers are allowed to participate in the operation

output		
RESULT :	ANY_PLAIN	the result of the operation; the type of the result depends on the types of the involved operands; as above, the same rules of the "XOR" (^) operator apply (see [1] for details) ⇒ - if only bitstrings are used, the result is a bitstring of the larger involved size; - if also unsigned integers are used, the result is an unsigned integer of the larger involved size

NOT

Negates (bitwise) the value of the given operand.

RESULT = **NOT** (v)

Same as:
`!V`

input		
v :	ANY_PLAIN	the value to be negated; the limitations are the same as the "!" operator (see [1] for details)

output		
RESULT :	ANY_PLAIN	the result of the negation;

the type of the result depends on the type of the operand (usually but not necessarily the same);
as above, the same rules of the "!" operator apply (see [1] for details)

Comparison functions

A whole family of functions is dedicated to the comparisons of multiple values. All of them work in a similar way: they accept as input a list of values (in variable number) and apply the same comparison operator to all. The output is always a boolean value stating the result of the comparison.

RESULT = **<XX>** (V1, V2 [, … , VN])

input

V# :	ANY_PLAIN	the list of values to be compared; values of heterogeneous types are allowed to be given; for compatibility between different types refer to [1] and the notes about the corresponding comparison operators (<, >, <=, >=, =, <>)

output

RESULT :	BOOL	the result of the comparison

See the following descriptions for notes about the individual functions.

LT

"Less Than", applied to all the operands.
TRUE if the parameters are given in a monotonic strictly increasing sequence.
Same as:

```
((V1 < V2) AND (V2 < V3) AND (V3 < V4) AND …)
```

LE

"Less than or Equal to", applied to all the operands.
TRUE if the parameters are given in a monotonic non-decreasing sequence.
Same as:

```
((V1 <= V2) AND (V2 <= V3) AND (V3 <= V4) AND …)
```

GT

"Greater Than", applied to all the operands.
TRUE if the parameters are given in a monotonic strictly decreasing sequence.
Same as:

```
((V1 > V2) AND (V2 > V3) AND (V3 > V4) AND …)
```

GE

"Greater than or Equal to", applied to all the operands.
TRUE if the parameters are given in a monotonic non-increasing sequence.
Same as:

```
((V1 >= V2) AND (V2 >= V3) AND (V3 >= V4) AND …)
```

EQ

"Equal to", applied to all the operands.
TRUE if all the parameters are equal.
Same as:

```
((V1 = V2) AND (V2 = V3) AND (V3 = V4) AND …)
```

NE

"Not Equal to", applied to all the operands.
TRUE if all the parameters are different.
Same as:

```
(((V1 <> V2) AND (V1 <> V3) AND (V1 <> V4) AND …) AND
 ((V2 <> V3) AND (V2 <> V4) AND (V2 <> V5) AND …) AND
```

```
((V3 <> V4) AND (V3 <> V5) AND (V3 <> V6) AND …) AND …)
```

String functions

LEN

Returns the length, in characters, of a given string.

LENGTH = **LEN** (STRVAL)

input		
STRVAL :	ANY_STRING	the interested string

output		
LENGTH :	UINT	number of characters in the string; not to be confused with the allocated size of the string

LEFT

Creates a string made of the first (left) N characters of another given one.

RESULT = **LEFT** (STRVAL, LENGTH)

input		
STRVAL :	ANY_STRING	the initial string, from which the result is extracted
LENGTH :	ANY_INT	the number of characters to extract; this is the maximum length of the RESULT string; the RESULT could be shorter than this if not enough characters exist in the source STRVAL

output		
RESULT :	ANY_STRING	the sub-string extracted from the source STRVAL; will replicate the exact type of the source string

RIGHT

Creates a string made of the last (right) N characters of another given one.

RESULT = **RIGHT** (STRVAL, LENGTH)

input		
STRVAL :	ANY_STRING	the initial string, from which the result is extracted
LENGTH :	ANY_INT	the number of characters to extract; this is the maximum length of the RESULT string; the RESULT could be shorter than this if not enough characters exist in the source STRVAL

output		
RESULT :	ANY_STRING	the sub-string extracted from the source STRVAL; will replicate the exact type of the source string

MID

Creates a string made of a given number of characters extracted from another given one, starting from a given position.

RESULT = **MID** (STRVAL, LENGTH, POSITION)

Example:
```
MID ("abcdef", 2, 3)   = "cd"
MID ("abcdef", 2, -10) = "ab"
MID ("abcdef", 2, 10)  = ""
```

input

STRVAL :	ANY_STRING	the initial string, from which the result is extracted
LENGTH :	ANY_INT	the number of characters to extract; this is the maximum length of the RESULT string; the RESULT could be shorter than this if not enough characters exist in the source STRVAL
POSITION :	ANY_INT	the index (base 1) of the 1st character of the source string, starting from which the sub-string has to be extracted; if this index is < 1, then the extraction starts from the 1st character of the source string; if this index exceeds the actual length of the source, the result will simply be an empty string

output

RESULT :	ANY_STRING	the sub-string extracted from the source STRVAL; will replicate the exact type of the source string

CONCAT

Creates a string made as the concatenation of a number of other given strings.

RESULT = **CONCAT** (STR1, STR2 [..., STRN])

Same as:
```
STR1 + STR2 + … STRN
```
Example:
```
CONCAT ("ab", "cd", "ef") = "abcdef"
```

input

STR# :	ANY_STRING	all the strings that have to be concatenated

output

RESULT :	ANY_STRING	the complete concatenated string; the type will be the higher class of the involved source strings (WSTRING > STRING)

INSERT

Creates a string inserting one in another, starting from a given position.

RESULT = **INSERT** (STR1, STR2, POSITION)

Example:
```
INSERT ("abcd", "1234", 3)   = "ab1234cd"
INSERT ("abcd", "1234", -10) = "1234abcd"
INSERT ("abcd", "1234", 10)  = "abcd1234"
```

input

STR1 :	ANY_STRING	the string where STR2 will be inserted
STR2 :	ANY_STRING	the string that will be inserted in STR1
POSITION :	ANY_INT	the index (base 1) of the 1st character of STR2 when inserted in STR1; if this index is < 1, then the insertion starts from the 1st character of the source string; if this index exceeds the actual length of the source, then the insertion starts from the end of the string (working as a plain append)

output

RESULT :	ANY_STRING	the complete concatenated string; the type will be the higher class of the involved source strings (WSTRING > STRING)

DELETE

Creates a string deleting characters from another given one.

RESULT = **DELETE** (STRVAL, LENGTH, POSITION)

Example:

```
DELETE ("abcdef", 2, 3)   = "abef"
DELETE ("abcdef", 2, -10) = "cdef"
DELETE ("abcdef", 2, 10)  = "abcdef"
```

input

STRVAL :	ANY_STRING	the initial string, from which characters are deleted
LENGTH :	ANY_INT	the number of characters that have to be deleted from STRVAL
POSITION :	ANY_INT	the index (base 1) of the 1st character of STRVAL that has to be deleted if this index is < 1, then the deletion starts from the 1st character of the source string; if this index exceeds the actual length of the source, then nothing will be deleted

output

RESULT :	ANY_STRING	the final string, purged from unneeded characters; will replicate the exact type of the source string

REPLACE

Creates a string replacing some characters of one with characters of another.

RESULT = **REPLACE** (STR1, STR2, LENGTH, POSITION)

Example:

```
REPLACE ("abcdef", "1234", 2, 3)   = "ab12ef"
REPLACE ("abcdef", "1234", 4, -10) = "1234ef"
REPLACE ("abcdef", "1234", 4, 10)  = "abcdef"
REPLACE ("abcdef", "1234", 4, 5)   = "abcd12"
```

input

STR1 :	ANY_STRING	the string that will have some characters replaced
STR2 :	ANY_STRING	the string that provides the characters to be used in the replacement
LENGTH :	ANY_INT	the number of characters that have to be replaced;

			this is actually a maximum number of characters, since in STR2 there could be less characters than the declared needed ones; in this case only the available characters are replaced, while the others remain unaffected;
POSITION :	ANY_INT		the index (base 1) of the 1st character of STR1 that has to be replaced; if this index is < 1, then the replacement starts from the 1st character of the source string; if this index exceeds the actual length of the source, then nothing will be replaced; if the replacement starts within the actual source limits, but (due to POSITION + LENGTH) ends after its available length, then only the available characters are replaced (no new characters are appended to the initial string: the result is always expected to have the same length as the source STR1)

output

RESULT :	ANY_STRING		the modified string; the type will be the higher class of the involved source strings (WSTRING > STRING)

REVERSE

Creates a string inverting the position of all of its characters.

RESULT = **REVERSE** (STRVAL)

Example:
```
REVERSE ("abcdef") = "fedcba"
```

input

STRVAL :	ANY_STRING		the string that has to be reversed

output

RESULT :	ANY_STRING		the reversed string; will replicate the exact type of the source string

SPLIT

Splits a string formatted with separated pieces, and produces an array containing a string piece in each of its elements.

ARRAY = **SPLIT** (STRVAL [, SEPARATOR [, ELEMENTS [, START]]])

input

STRVAL :	ANY_STRING		the string that has to be split
SEPARATOR :	ANY_CHARS	[OPTIONAL]	the character that has to be recognized as pieces separator; can be given as any kind of character or any kind of string; in case of strings, only the 1st character is considered; **SYSBLANKS** ("$FFFF") is a special character used to indicate that all sequences of spaces (0x20) and tabs (0x09) must be considered a separation; in this case sequences of consecutive blanks (consecutive multiple spaces and/or tabs) only count for a single separator if missing, the default separator is **SYSBLANKS**
ELEMENTS :	ANY_INT	[OPTIONAL]	the number of pieces returned in the output array; must be ≥ 0: 0 means all the elements must be returned, > 0 is used to limit the output size;

			if missing, the function returns all the pieces found in the given string; if more elements than the available pieces are requested, then a number of array elements might be returned as empty strings
START :	ANY_INT	[OPTIONAL]	the position index (base 1) of the 1st string piece returned in the output array; must be > 0; if missing, the function starts from the 1st string piece; if the given index is greater than the number of pieces available in the given string, then all the returned elements will be empty; even if the starting position is within the number of existing pieces, it's still possible that START + ELEMENTS exceeds the available range; in this case a number of elements in the output array might be returned as empty string

output

ARRAY :	ANY	the returned value is actually an array of strings (same type of strings as the source STRVAL); the number of elements will depend on the given ELEMENTS parameter, or on the actual number of pieces found in the source STRVAL

Examples:

- generic application cases:

```
SPLIT ("ab cd ef") = ("ab" , "cd" , "ef")
SPLIT ("ab   cd    ef") = ("ab" , "cd" , "ef")
SPLIT ("ab   cd    ef", SYSBLANKS) = ("ab" , "cd" , "ef")
SPLIT ("ab cd ef", " ") = ("ab" , "cd" , "ef")
SPLIT ("ab  cd  ef", " ") = ("ab" , "" , "cd" , "" , "ef")
```

- watch out for proper separator:

```
SPLIT ("ab cd ef", ";") = ("ab cd ef")
```

- watch out for multiple occurrences of a separator:

```
SPLIT ("ab;;cd;ef", ";", 3) = ("ab" , "" , "cd")
```

- only few pieces requested:

```
SPLIT ("ab,cd,ef", ",", 2) = ("ab" , "cd")
```

- moving forward the extraction point:

```
SPLIT ("ab,cd,ef,gh", ",", 2, 3) = ("ef" , "gh")
```

- requesting more pieces than available:

```
SPLIT ("ab,cd,ef", ",", 5) = ("ab" , "cd" , "ef" , "" , "")
```

- requesting more pieces than available from a given position:

```
SPLIT ("ab cd ef gh ij", " ", 3, 4) = ("gh" , "ij" , "")
```

- when the extraction begins past the end of the string:

```
SPLIT ("ab cd ef gh ij", " ", 3, 6) = ("" , "" , "")
```

A note about the assignment of the returned array to a result variable: the standard rules for assignments of arrays apply, so programmers should take care of the following:

- the type of the variable array elements must be exactly the same as the type of the returned elements; this means there must be correspondence between STRING/STRING and WSTRING/WSTRING types;
- the variable array is allowed to have a different number of elements than the returned array only in case of LAX_TYPES option enabled;
- in this case, having a bigger variable will leave some of its elements empty, while having a smaller variable will only make it able to store the first pieces.

FIND

Finds the position of the occurrence of a string within another.

POSITION = **FIND** (STR1, STR2)

Example:

```
FIND ("abCDCDef", "CD") = 3
```

input

STR1 :	ANY_STRING	the string that might contain STR2
STR2 :	ANY_STRING	the sub-string that might be contained in STR1

output

POSITION :	UINT	the position of the (first) occurrence of STR2 within STR1; this is basically the index (base 1) of the 1st character of STR2 in STR1; if the sub-string is not found, the result is 0; the performed search is case sensitive

RFIND

Finds the position of the occurrence of a string within another;
the search of the sub-string is performed starting from the right.

POSITION = **RFIND** (STR1, STR2)

Example:
```
RFIND ("abCDCDef", "CD") = 5
```

input

STR1 :	ANY_STRING	the string that might contain STR2
STR2 :	ANY_STRING	the sub-string that might be contained in STR1

output

POSITION :	UINT	the position of the (last) occurrence of STR2 within STR1; this is basically the index (base 1) of the 1st character of STR2 in STR1; if the sub-string is not found, the result is 0; the performed search is case sensitive

LCASE

Converts a string in lowercase characters.

RESULT = **LCASE** (STRVAL)

Example:
```
LCASE ("AbCdEf") = "abcdef"
```

input

STRVAL :	ANY_STRING	the string that has to be converted

output

RESULT :	ANY_STRING	a copy of the source string with all the alphabetic characters converted in lowercase; will replicate the exact type of the source string

UCASE

Converts a string in uppercase characters.

RESULT = **UCASE** (STRVAL)

Example:
```
UCASE ("AbCdEf") = "ABCDEF"
```

input

STRVAL :	ANY_STRING	the string that has to be converted

output

RESULT :	ANY_STRING	a copy of the source string with all the alphabetic characters converted in uppercase; will replicate the exact type of the source string

TRIM

Removes from the head (left) and from the tail (right) of a string all the consecutive occurrences of a given character.

RESULT = **TRIM** (STRVAL, CHARACTER)

Example:
```
TRIM ("---abcdef---", "-") = "abcdef"
```

input

STRVAL :	ANY_STRING		the string that has to be trimmed
CHARACTER :	ANY_CHARS	[OPTIONAL]	the character that has to be removed from the source string; can be given as any kind of character or any kind of string; in case of strings, only the 1st character is considered; the special constant SYSBLANKS ("$FFFF") is supported, meaning that all spaces (0x20) and tabs (0x09) must be removed; if missing, the default is SYSBLANKS

output

RESULT :	ANY_STRING	the source string purged from all the initial and final occurrences of the character; will replicate the exact type of the source string

LTRIM

Removes from the head (left) of a string all the consecutive occurrences of a given character.

RESULT = **LTRIM** (STRVAL, CHARACTER)

Example:
```
LTRIM ("---abcdef---", "-") = "abcdef---"
```

input

STRVAL :	ANY_STRING		the string that has to be trimmed
CHARACTER :	ANY_CHARS	[OPTIONAL]	the character that has to be removed from the source string; can be given as any kind of character or any kind of string; in case of strings, only the 1st character is considered; the special constant SYSBLANKS ("$FFFF") is supported, meaning that all spaces (0x20) and tabs (0x09) must be removed; if missing, the default is SYSBLANKS

output

RESULT :	ANY_STRING	the source string purged from all the initial occurrences of the character; will replicate the exact type of the source string

RTRIM

Removes from the tail (right) of a string all the consecutive occurrences of a given character.

RESULT = **RTRIM** (STRVAL, CHARACTER)

Example:
```
RTRIM ("---abcdef---", "-") = "---abcdef"
```

input

STRVAL :	ANY_STRING		the string that has to be trimmed
CHARACTER :	ANY_CHARS	[OPTIONAL]	the character that has to be removed from the source string; can be given as any kind of character or any kind of string; in case of strings, only the 1st character is considered; the special constant **SYSBLANKS** ("$FFFF") is supported, meaning that all spaces (0x20) and tabs (0x09) must be removed; if missing, the default is **SYSBLANKS**

output

RESULT :	ANY_STRING	the source string purged from all the final occurrences of the character; will replicate the exact type of the source string

SETLENGTH

Sets a precise length for a string.

RESULT = **SETLENGTH** (STRVAL, FILLCHAR, LENGTH, RIGHT)

Example:
```
SETLENGTH ("abcdef", "-", 3, TRUE)  = "abc"       // cut
SETLENGTH ("abcdef", "-", 8, FALSE) = "--abcdef"  // extend on the left
SETLENGTH ("abcdef", "-", 8, TRUE)  = "abcdef--"  // extend on the right
SETLENGTH ("abcdef", "",  8, TRUE)  = "abcdef"    // reallocate but don't change
```

input

STRVAL :	ANY_STRING	the string that has to be changed
FILLCHAR :	ANY_CHARS	the character that has to be used to fill up the string space in case the given LENGTH is longer than the original STRVAL length; can be given as any kind of character or any kind of string; in case of strings, only the 1st character is used
LENGTH :	ANY_INT	new length for the string; if shorter than the original length, the source string is truncated; if longer than the original length, the source string is padded with FILLCHAR characters
RIGHT :	BOOL	declares where the padding has to happen in case of longer length; if TRUE, FILLCHARs are appended on the right, otherwise on the left

output

RESULT :	ANY_STRING	the source string with the length adjusted; will replicate the exact type of the source string

HEX

Formats an integer value in a string with hexadecimal notation.

RESULT = **HEX** (V)

Example:
```
HEX (171) = "AB"
```

input

V :	ANY_INT	the value that has to be formatted

output

RESULT :	STRING	the string where the value V is written in hexadecimal

OCT

Formats an integer value in a string with octal notation.

RESULT = **OCT** (V)

Example:
```
OCT (171) = "253"
```

input

V :	ANY_INT	the value that has to be formatted

output

RESULT :	STRING	the string where the value V is written in octal

BIN

Formats an integer value in a string with binary notation.

RESULT = **BIN** (V)

Example:
```
BIN (171) = "10101011"
```

input

V :	ANY_INT	the value that has to be formatted

output

RESULT :	STRING	the string where the value V is written in binary

ASC

Retrieves the (ascii/unicode) code of a character.

RESULT = **ASC** (CHARACTER)

Example:
```
ASC ("A") = 65
```

input

CHARACTER :	ANY_CHARS	the interested character; can be given in character or string form; in case of strings, the 1st character is considered; in case of character type, this function acts the same way as an **ANY_TO_INT**; in case of strings though, an **ANY_TO_INT** would return the numeric value formatted in the string, while this **ASC** function returns the code of its first character (similar to an **ANY_TO_INT**(STRING[0]))

output

RESULT :	UINT	the code of the character

Utility functions

ISSTRUCTURE

Checks a value to see if its type is a derived structure.

STATE = **ISSTRUCTURE** (V)

input		
V :	ANY	the variable/value to be checked

output		
STATE :	BOOL	returns TRUE if the submitted value is a structure; FALSE otherwise

ISFUNCTION

Checks a value to see if it is a function block instance.

STATE = **ISFUNCTION** (V)

input		
V :	ANY	the variable/value to be checked

output		
STATE :	BOOL	returns TRUE if the submitted value is defined as the instance of a function block; FALSE otherwise

ISARRAY

Checks a value to see if its type is an array.

STATE = **ISARRAY** (V)

input		
V :	ANY	the variable/value to be checked

output		
STATE :	BOOL	returns TRUE if the submitted value is an array (of whatever type); FALSE otherwise

NUMDIM

Counts the number of dimensions of an array.

DIMENSIONS = **NUMDIM** (V)

input		
V :	ANY	the variable/value to be checked; ideally, but not necessarily, an array

output

DIMENSIONS :	ULINT	returns the number of dimensions of the submitted array; if v is not an array, 0 is returned

LBOUND

Retrieves the lower bound of the range of indexes of an array.

INDEX = **LBOUND** (V)

Example:

for an array like	`VAR A1[2..5] OF INT; END_VAR;`	the bound is	`LBOUND(A1) = 2`
for an array like	`VAR A2[7] OF INT; END_VAR;`	the bound is	`LBOUND(A2) = 0`

input

V :	ANY	the variable/value to be checked; ideally, but not necessarily, an array

output

INDEX :	LINT	returns the lowest possible index (lower bound) for the elements of the submitted array; if v is not an array, the function returns 0; in case of array with multiple dimensions, only the 1st dimension is considered

Note that to handle multiple dimensions, ISARRAY and NUMDIM can be used to trace the correct conditions, and calls like the following can be used:

```
VarLb1 = LBOUND (V);                  // lower bound of 1st dimension
VarLb2 = LBOUND (V[VarLb1]);          // lower bound of 2nd dimension
VarLb3 = LBOUND (V[VarLb1,VarLb2]);   // lower bound of 3rd dimension
```

and so on.

UBOUND

Retrieves the upper bound of the range of indexes of an array.

INDEX = **UBOUND** (V)

Example:

for an array like	`VAR A1[2..5] OF INT; END_VAR;`	the bound is	`UBOUND(A1) = 5`
for an array like	`VAR A2[7] OF INT; END_VAR;`	the bound is	`UBOUND(A2) = 6`

input

V :	ANY	the variable/value to be checked; ideally, but not necessarily, an array

output

INDEX :	LINT	returns the highest possible index (upper bound) for the elements of the submitted array; if v is not an array, the function returns 0; in case of array with multiple dimensions, only the 1st dimension is considered; see notes above

SIZEOF

Retrieves the total size, in bytes, of the value of the given parameter.

Mainly used to measure the size of complex types, such as user-defined structures or arrays, or nidified components.

SIZE = **SIZEOF** (V)

Examples:

for an array like
```
VAR
  VA[2..5] OF INT;
END_VAR;
```
the size would be
```
SIZEOF(VA) = 8
```

for a structure like
```
TYPE StrA :
   STRUCT
      f1 : SINT;
      f2 : SINT;
   END_STRUCT;
END_TYPE;
TYPE StrB :
   STRUCT
      f1 : DINT;
      f2 : ARRAY [4] OF StrA;
   END_STRUCT;
END_TYPE;
VAR
  VS : StrB;
END_VAR;
```
the size would be
```
SIZEOF(VS) = 12
SIZEOF(VS.f2) = 8
SIZEOF(VS.f2[3].f1) = 1
```

input

V :	ANY	the variable/value to be checked

output

SIZE :	LINT	returns the size, in bytes, of the value of the given parameter (a measure of the memory space taken by the value)

TYPEOF

Retrieves the name of the type of the given parameter.
Mainly used for debug purposes, in cases where the parameter comes from sources with variable type (such as system functions with output in the ANY class.

TYPE = **TYPEOF** (V)

Example:

```
TYPE MyTypeStruct :
   STRUCT FieldName : INT; END_STRUCT;
END_TYPE;
TYPE MyTypeArray :
   ARRAY [1..10] OF ARRAY [3..4] OF INT;
END_TYPE;
TYPE MyTypeEnum :
   ( val1, val2, val3 ) := val2 ;
END_TYPE ;

VAR
```

```
    si: INT;
    au : ARRAY [5] OF UDINT;
    ca : MyTypeArray;
    cs : MyTypeStruct;
    ce : MyTypeEnum;
    aca : ARRAY[2] OF MyTypeArray;
    acs : ARRAY[2] OF MyTypeStruct;
    ace : ARRAY[2] OF MyTypeEnum;
  END_VAR;

  _TRACE ( TYPEOF (123) );
  _TRACE ( TYPEOF (si) );
  _TRACE ( TYPEOF (au) );
  _TRACE ( TYPEOF (ca) );
  _TRACE ( TYPEOF (cs) );
  _TRACE ( TYPEOF (ce) );
  _TRACE ( TYPEOF (aca) );
  _TRACE ( TYPEOF (acs) );
  _TRACE ( TYPEOF (ace) );
```

the output would be:

```
  ULINT
  INT
  ARRAY [5:0..4] OF UDINT
  ARRAY [20:1..10,3..4] OF INT
  MyTypeStruct (STRUCT)
  MyTypeEnum (ENUM)
  ARRAY [40:0..1,1..10,3..4] OF INT
  ARRAY [2:0..1] OF MyTypeStruct (STRUCT)
  ARRAY [2:0..1] OF MyTypeEnum (ENUM)
```

as shown:
- either standard or custom type names can be retrieved,
- complex types show their class (struct, enum, function),
- arrays show their total number of elements, along with the exact ranges of all their dimensions

input

V :	ANY	the variable/value to be checked

output

TYPE :	WSTRING	returns the name of the parameter data type

Value conversion functions

DEG_TO_RAD

Converts an angle measure from degrees to radians.

RAD = **DEG_TO_RAD** (DEG)

Example:
```
DEG_TO_RAD(180) = _PI
```

input		
DEG :	ANY_NUM	the measure of an angle in degrees

output		
RAD :	LREAL	returns the measure converted in radians

RAD_TO_DEG

Converts an angle measure from radians to degrees.

DEG = **RAD_TO_DEG** (RAD)

Example:
```
RAD_TO_DEG(_PI) = 180
```

input		
RAD :	ANY_NUM	the measure of an angle in radians

output		
DEG :	LREAL	returns the measure converted in degrees

BCD_TO_BIN

Converts an unsigned integer value coded in BCD in a plain binary value.

BIN = **BCD_TO_BIN** (BCD)

Example:
```
BCD_TO_BIN(16#1234) = 16#4D2 = 1234
```

input		
BCD :	ANY_UNSIGNED	a value coded in BCD (the binary content of the value must be coded in BCD)

output		
BIN :	ANY_UNSIGNED	the value converted from the original BCD to its plain binary value; replicates the type of the submitted BCD

BIN_TO_BCD

Converts any plain unsigned integer value in its corresponding BCD coding.

BCD = **BIN_TO_BCD** (BIN)

Example:
```
BIN_TO_BCD(1234) = 16#1234 = 4660
```

input

BIN :	ANY_UNSIGNED	the value to be converted in BCD

output

BCD :	UI64	the value converted in BCD; note that the size of the result (64 bits) could be unable to fit the entire converted values; in that case truncations might happen

Type conversion functions

A whole family of functions is dedicated to the conversion of the values types: a function **ANY_TO_<TYPE>** exists for each of the existing elementary types, and all of them work in a similar way: they accept as input a value of any of the existing 'plain' types (usually any type except arrays and structures), and return it converted in a value of the selected type.

CONVERTEDVAL = **ANY_TO_<TYPE>** (SOURCEVAL)

input		
SOURCEVAL :	ANY_PLAIN	the value to be converted; from function to function, differences might exist among the list of types actually allowed in input

output		
CONVERTEDVAL :	<TYPE>	the result of the conversion

See the following descriptions for notes about the individual functions.

ANY_TO_SINT

Convert any plain value to a signed short integer (8 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the lowest byte of the result is retained ad result.
Can be invoked with the alias **CSINT**.

ANY_TO_INT

Convert any plain value to a signed integer (16 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the 2 lowest bytes of the result is retained ad result.
Can be invoked with the alias **CINT**.

ANY_TO_DINT

Convert any plain value to a signed double integer (32 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the 4 lowest bytes of the result is retained ad result.
Can be invoked with the alias **CDINT**.

ANY_TO_LINT

Convert any plain value to a signed long integer (64 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Can be invoked with the alias **CLINT**.

ANY_TO_USINT

Convert any plain value to an unsigned short integer (8 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the lowest byte of the result is retained ad result.
Can be invoked with the alias **CUSINT**.

ANY_TO_UINT

Convert any plain value to an unsigned integer (16 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the 2 lowest bytes of the result is retained ad result.
Can be invoked with the alias **CUINT**.

ANY_TO_UDINT

Convert any plain value to an unsigned double integer (32 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Range overflows in the source value are not considered errors: simply the 4 lowest bytes of the result is retained ad result.
Can be invoked with the alias **CUDINT**.

ANY_TO_ULINT

Convert any plain value to an unsigned long integer (64 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Can be invoked with the alias **CULINT**.

ANY_TO_REAL

Convert any plain value to a single precision floating-point (32 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Can be invoked with the alias **CREAL**.

ANY_TO_LREAL

Convert any plain value to a double precision floating-point (64 bits).
. In case of strings, the written value is converted.
. In case of characters, their numeric code is converted.
. In case of times, dates and bitstrings, their pseudo-numeric value is converted.
Can be invoked with the alias **CLREAL**.

ANY_TO_TIME

Convert any plain value in a TIME value.
. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 32 bits integer). Range truncations might occur.
. LTIMEs are more precisely converted: other than being truncated if needed, their value of course loses the precision beyond the milliseconds.
. Strings are expected to have the same format as the TIME and LTIME constants of the language (for example "11d22h33m44s55ms", "11d22h33m44s55ms66us77ns").
Can be invoked with the alias **CTIME**.

ANY_TO_LTIME

Convert any plain value in a Long TIME value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 64 bits integer). Range truncations might occur.
. TIMEs are more precisely converted: they are simply translated into the new format, since both range and precision don't need adjustments.
. Strings are expected to have the same format as the TIME and LTIME constants of the language (for example "11d22h33m44s55ms", "11d22h33m44s55ms66us77ns").

Can be invoked with the alias **CLTIME**.

ANY_TO_DATE

Convert any plain value in a DATE value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 16 bits integer). Range truncations might occur.
. DTs and LDTs (date and time) have their date component taken and assigned to the result.
. Strings are expected to have the same format as the DATE constants of the language (for example "2018-5-24").

Can be invoked with the alias **CDATE**.

ANY_TO_TOD

Convert any plain value in a TIME_OF_DAY value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 32 bits integer). Range truncations might occur.
. DTs and LDTs (date and time) have their time component taken and assigned to the result; in case of LDTs, the precision might have to be truncated.
. LTODs are downsized to TODs, having their precision truncated.
. Strings are expected to have the same format as the TOD or LTOD constants of the language (for example "23:59:59.990", "23:59:59.990_234_075", "23:59:59.123456789").

Can be invoked with the alias **CTOD**.

ANY_TO_LTOD

Convert any plain value in a Long TIME_OF_DAY value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 64 bits integer). Range truncations might occur.
. DTs and LDTs (date and time) have their time component taken and assigned to the result; range and precision don't need adjustments.
. TODs are simply translated into the new format; again, range and precision don't need adjustments
. Strings are expected to have the same format as the TOD or LTOD constants of the language (for example "23:59:59.990", "23:59:59.990_234_075", "23:59:59.123456789").

Can be invoked with the alias **CLTOD**.

ANY_TO_DT

Convert any plain value in a DATE_AND_TIME value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 32 bits integer). Range truncations might occur.
. DATEs are used to compose a DT with a time component set to 00:00:00.0.
. LDTs are downsized to DTs, having the precision of the time component truncated.
. Strings are expected to have the same format as the DT or LDT constants of the language (for example "2018-5-24-23:59:59.990", "2018-5-24-23:59:59.990_234_075").

Can be invoked with the alias **CDT**.

ANY_TO_LDT

Convert any plain value in a Long DATE_AND_TIME value.

. All numeric and pseudo-numeric values, including characters, bitstrings and incompatible dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 64 bits integer). Range truncations might occur.
. DATEs are used to compose a LDT with a time component set to 00:00:00.0.
. DTs are simply converted to LDTs, with no precision issues.
. Strings are expected to have the same format as the DT or LDT constants of the language (for example "2018-5-24-23:59:59.990", "2018-5-24-23:59:59.990_234_075").

Can be invoked with the alias **CLDT**.

ANY_TO_STRING

Convert any plain value in an 8 bits string.

. All numeric values, both integer and floating-point, are simply formatted in the corresponding string.
. CHARs and WCHARs are converted in a string made of 1 only character; wide chars might have their code truncated.
. WSTRINGs are converted in corresponding strings with characters truncated to 8 bits codes.
. All the date and time values (TIME, LTIME, DATE, TOD, LTOD, DT, LDT) are formatted in strings with the same format as the corresponding constant of the language.

Can be invoked with the alias **CSTRING**.

ANY_TO_WSTRING

Convert any plain value in a 16 bits string.

. All numeric values, both integer and floating-point, are simply formatted in the corresponding wide string.
. CHARs and WCHARs are converted in a wide string made of 1 only character.
. STRINGs are simply copied in corresponding wide strings.
. All the date and time values (TIME, LTIME, DATE, TOD, LTOD, DT, LDT) are formatted in wide strings with the same format as the corresponding constant of the language.

Can be invoked with the alias **CWSTRING**.

ANY_TO_CHAR

Convert any plain value in an 8 bits character.

. All numeric and pseudo-numeric values, including bitstrings, dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 8 bits integer), that represents the code (ascii) of the character. Range truncations might occur.
. Strings are converted in a result made of their first character only; wide strings might have the character code truncated; empty strings return NUL (0).

Can be invoked with the alias **CCHAR**.

ANY_TO_WCHAR

Convert any plain value in a 16 bits character.

. All numeric and pseudo-numeric values, including bitstrings, dates and times, have simply their numeric value converted into the numeric value of the result (unsigned 16 bits integer), that represents the code (unicode) of the character. Range truncations might occur.
. Strings are converted in a result made of their first character only; empty strings return NUL (0).

Can be invoked with the alias **CWCHAR**.

ANY_TO_BOOL

Convert any plain value in a boolean.

. All numeric and pseudo-numeric values, including other bitstrings, characters, dated and times, make a FALSE boolean value if their value is 0; they make a TRUE boolean instead for any value different from 0.
. In case of strings, they are first converted in integers; the numeric values then are converted as above.

Can be invoked with the alias **CBOOL**.

ANY_TO_BYTE

Convert any plain value in a byte (8 bits bitstring).

. All numeric and pseudo-numeric values, including other bitstrings, characters, dated and times, have simply their numeric value converted into the numeric value of the result (unsigned 8 bits integer). Range truncations might occur.
. In case of strings, their formatted content is converted (and truncated if needed).

Can be invoked with the alias **CBYTE**.

ANY_TO_WORD

Convert any plain value in a word (16 bits bitstring).

. All numeric and pseudo-numeric values, including other bitstrings, characters, dated and times, have simply their numeric value converted into the numeric value of the result (unsigned 16 bits integer). Range truncations might occur.
. In case of strings, their formatted content is converted (and truncated if needed).

Can be invoked with the alias **CWORD**.

ANY_TO_DWORD

Convert any plain value in a double word (32 bits bitstring).

. All numeric and pseudo-numeric values, including other bitstrings, characters, dated and times, have simply their numeric value converted into the numeric value of the result (unsigned 32 bits integer). Range truncations might occur.
. In case of strings, their formatted content is converted (and truncated if needed).

Can be invoked with the alias **CDWORD**.

ANY_TO_LWORD

Convert any plain value in a long word (64 bits bitstring).

. All numeric and pseudo-numeric values, including other bitstrings, characters, dated and times, have simply their numeric value converted into the numeric value of the result (unsigned 64 bits integer).
. In case of strings, their formatted content is converted.

Can be invoked with the alias **CLWORD**.

< CONSTANTS >

The following symbolic keywords can be used as constant values.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding values were written in their place.

symbolic	type	value
FALSE	BOOL	0
TRUE	BOOL	1
_PI	LREAL	3.14159265358979323846
_E	LREAL	2.71828182845904523536

4. SYSTEM

TRACE

For debugging purposes, used to trace a message on few output channels available to the system.

TRACE (MESSAGE)

input		
MESSAGE :	ANY_STRING	message string to be traced; maximum length fixed at 10240 characters

The given message will be:
- printed on the local runtime process stdout stream;
- sent to ST debuggers attached from the configurator environment;
- sent to attached Visual Studio debuggers (for developers).

RUNAPPLICATION

Executes an application with given executable name and command-line parameters.

RESULT = **RUNAPPLICATION** (NAME, PARAMETERS, MODE [, STDOUT [, STDERR]])

input

NAME :	ANY_STRING	path and name of the executable file (path INcluded); the path can be either relative or absolute, and is allowed to start with system keys such as $RESOURCES or $LOG (see 'File paths conventions' at the beginning of the 'FILE' section)
PARAMETERS :	ANY_STRING	command line parameters; includes all the extra parameters to be passed to the process; programmers can quote parameters where separation ambiguities may arise; can be an empty string if no parameters are required
MODE :	ANY_INT	defines the behaviour of the runtime in its interaction with the created process; supported codes are: **0** (SYSRUNASYNCUNIQUE) : the new process runs asynchronously, but the runtime makes sure to have only one external application running; typically used when applications with a user interface have to be executed and the system doesn't allow to switch between them; **1** (SYSRUNASYNCFREE) : the new process runs asynchronously; the runtime just creates and starts it; no further checks are involved; programmers are allowed to execute as many processes as they need this way; **2** (SYSRUNBLOCKING) : the new process is 'blocking' for the script; the function will not return until the new process has terminated; of course one only blocking process at a time can exist
STDOUT :	ANY_STRING	[OPTIONAL] an optional file name; if given, all the standard output stream of the new process will be redirected to this file
STDERR :	ANY_STRING	[OPTIONAL] an optional file name; if given, all the standard error stream of the new process will be redirected to this file

output

RESULT :	UDINT	application result (the application process exit code); this output is meaningful only if the application has been executed in SYSRUNBLOCKING mode (the runtime could wait for the application to actually give its result); in all other modes the returned code is 0

About the optional STDOUT and STDERR files that can be used to redirect the standard output streams:
- these files are optional; omitting both of them doesn't alter the standard output behaviour;
- if only a STDOUT is given (and no STDERR), then the standard output is redirected, while the standard error is discarded;
- each of them can be given as an empty string; in this case the empty name is implicitly used as an indication to redirect to the standard NULL device (just like giving an explicit "NUL" file in Windows, or a "/dev/null" file in Linux); in this case no redirection file is created, and the standard output is simply discarded by the system;
- both streams can be redirected to the same file, simply giving the same file name in both parameters;
- the given file names are expected to follow the conventions explained at the beginning of the 5. COMMON FILES chapter, with respect to the individuation of absolute, relative or system paths;
- the specified files are created as text files with UTF-8 encoding.

example

```
VAR
   i : INT;
   f : UDINT;
   s : WSTRING [256];
END_VAR;

// Execute 3 instances of a test application, each with its own output
FOR i := 1 TO 3 DO
   RUNAPPLICATION ( 'C:\[MyApps]\StdoutTest.exe', '', SYSRUNBLOCKING,
                    'C:\[Documents]\OUT' + ANY_TO_STRING (i) + '.TXT',
                    'C:\[Documents]\ERR' + ANY_TO_STRING (i) + '.TXT' );
END_FOR;

// Read the output of the 1st executed process and retrace it
f := FILE_OPEN ('C:\[Documents]\OUT1.TXT', FILEREAD, FILEDONTCREATE);
FILE_READENCODING (f);  // Acquire the UTF-8 marker
WHILE (FILE_ISEOF(f) = FALSE) DO
   s := FILE_READSTRING (f, 256);
   _TRACE (s);
END_WHILE;
FILE_CLOSE (f);
```

KILLAPPLICATION

Terminates an application currently in execution.

KILLAPPLICATION (NAME)

input

NAME :	ANY_STRING	simple name of the executable file (path EXcluded)

The system forcefully terminates a process originated by an executable file with the given name.
If multiple existing processes are linked to the same file, or to files with the same name, the system simply terminates the first process that satisfies the condition.

RUNSCRIPT

Executes a script code.

RUNSCRIPT (CODE)

input

CODE :	ANY_STRING	string containing a segment of ST script source code

Not to be confused with mere functions invocation.
This function accepts as parameter a whole segment of script code: it is basically meant to allow the execution of scripts dynamically created at runtime.

This function is meant for execution only; it is possible to feed it any kind of script code, BUT it is not possible:
- to create new variables, new types or define new functions;
- to invoke other **RUNSCRIPT** functions;
- to use _BREAK statements to force breakpoints for debuggers.

It's intended to be used in cases where it is not possible to create the code at design time (the code below is actually an example of what COULD instead be built at design time), or when the application itself demands that script pieces are provided at runtime only, or when debuggers are involved (debuggers are allowed to request the execution of lines of script code).

Keep in mind that the execution of code given through this function is much less performant than the execution of regular code prepared at design time.

example

```
VAR
   MV : INT;
   MV1, MV2, MV3 : INT;
END_VAR;

// Just do something…
RUNSCRIPT ('FOR MV := 1 TO 10 DO /* … do something … */; END_FOR;');

// Assign 123 to MV1, MV2 or MV3, depending on the result of the function
MV := SomeRandomFunction ();
IF (MV >= 1) AND (MV <= 3) THEN
   RUNSCRIPT ('MV' + ANY_TO_STRING (MV) + ' := 123;');
END_IF;

// The code above is doing something like the following
MV := SomeRandomFunction ();
CASE (MV)
   1 : MV1 := 123;
   2 : MV2 := 123;
   3 : MV3 := 123;
END_CASE;
```

EXITRUNTIME

Terminates the execution of the runtime.

EXITRUNTIME ()

The runtime is supposed to gracefully shut down, fully closing and releasing every resource currently in use, and flushing all pending buffers.

SLEEP
LSLEEP

Blocks the script execution for a given amount of time.

SLEEP (TIME)
LSLEEP (TIME)

input

TIME :	ANY_INT	duration of the wait; **SLEEP** : the parameter is given in milliseconds **LSLEEP** : the parameter is given in nanoseconds

ERRORGETMESSAGE

Retrieves a description message related to a given error code.

MESSAGE = **ERRORGETMESSAGE** (ERROR)

input		
ERROR :	ULINT	code of the error

output		
MESSAGE :	WSTRING	error description message

Unlike the variable **ERRMSG**, systematically giving the description of the error stored in **ERRNO**, this function can be used to retrieve descriptions of any existing error code.

ERRORGETMODULE

Retrieves the name of the runtime module associated to a given error code.

MODULE = **ERRORGETMODULE** (ERROR)

input		
ERROR :	ULINT	code of the error

output		
MODULE :	WSTRING	name of the runtime module

The returned name is supposed to indicate the runtime module that generated/detected the error.

ERRORRESET

Resets the system variable used to store execution errors information.

ERRORRESET ()

The function execution is equivalent to the instructions:
ERRNO := 0;
ERRMSG := '';

FLUSHCONFIG

Flushes on persistent storage the buffers of the platform system configuration store.

FLUSHCONFIG ()

On Windows the function is used to flush the registry;
on Linux it's a plain flush of the dedicated configuration store file.

FLUSHPERSISTENT

Flushes on persistent storage the buffers of all the runtime logs with pending data.

FLUSHPERSISTENT ()

The flush applies to log files of:
- persistent internal tags,
- persistent external tags,
- alarms history records,
- alarms statistics information,
- datalog samples buffers,
- scheduler tasks,
- FDA auditor records.

Many of the involved elements have their own dedicated script function as well.

REFRESHIPADDRESSES

Scans the ethernet adapters and updates the system variables with the list of the available addresses.

REFRESHIPADDRESSES ()

The function affects the following system tags:

"SYS_NumIpAddresses" :	the number of IP addresses currently available among all the existing ethernet adapters;
"SYS_IpAddresses" :	the list of the first 8 available IP addresses; there could be more than 1 address for each available ethernet adapter;
"SYS_EthNames" :	the list of the names of the ethernet adapters corresponding to the addresses set in the previous variable (there is a name in <SYS_EthNames> for each address in <SYS_IpAddresses>).

SETRESTAPIPREFIX

Sets a string used to identify the prefix in URLs of HTTP calls handled by the runtime web server.
Matching calls are the only ones forwarded to the dedicated events.

SETRESTAPIPREFIX (PREFIX)

input

PREFIX :	ANY_STRING	[OPTIONAL]	the prefix string used to identify the needed calls; - if a specific **string** is given, then only HTTP packets starting with it are forwarded to the events; - if an **empty** string is given, then all the packets are forwarded to the events; - if the parameter is **missing**, then the identification of the HTTP packets is disabled, and no packet is ever forwarded to the events

This function is used to allow the system to select the HTTP packets - among those received by the web server - that have to be forwarded to the dedicated "OnWebserverCall" runtime event functions.
The default behaviour at startup is: "nothing will be forwarded to the events". This is to avoid unwanted calls during the time between the startup and the first scripts execution (when an explicit initialization can be done with this function). If packets have to be forwarded to the events, then at least a call to this function is required.

HTTP calls to the web server are normally in the forms:

```
<ipaddress>:<port>
<ipaddress>:<port>/file
<ipaddress>:<port>/p/a/t/h/file
<ipaddress>:<port>/p/a/t/h/file?par1=val1&par2=val2
<ipaddress>:<port>/p/a/t/h/?par1=val1&par2=val2
<ipaddress>:<port>/?par1=val1&par2=val2
```

In general:

```
<ipaddress>:<port>[/path/][file][parameters]
```

This function allows the specification of a string segment that will have to match the beginning of the packet calls, following the `<ipaddress>:<port>` part (ideally but not necessarily in the "path" part).
For example, if a prefix like `api/v1/` were given,
then an URL string like `<ipaddress>:<port>/api/v1/status/Machine` would be forwarded to the events,
while an URL string like `<ipaddress>:<port>/js/master/RT.master.core.js` would not.

SETRESTAPIRESPONSE

Sets a response string to be sent back as answer to clients invoking managed REST API functions through the runtime web server.

SETRESTAPIRESPONSE (RESPONSE)

input

RESPONSE :	ANY_STRING	[OPTIONAL]	the response string that the web server will send back to the client that invoked the execution of a managed REST API function; - if a specific **string** is given (even empty), then that exact string will be used in the body of the HTTP answer - if the parameter is **missing**, then the answer is explicitly cleared: the webserver won't return any immediate answer to the caller (after the script execution) and proceed with extra management of the received packet (that could be interpreted as a file download request, as a standard ESA API request, and so on… any of the possible standard interpretation of ESA WebServer)

This function is meant to be used in scripts programmed to handle the execution of REST API functions.
A coherent setup would include:
- an active event OnWebserverCall;
- an ST script programmed on the event;
- the initialization of the prefix for the identification of the managed API calls (ideally invoked at startup, see **SETRESTAPIPREFIX**);
- the management of the API within the script;
- the preparation of the answer string to send to the caller (with this **SETRESTAPIRESPONSE**).

This function should only be called by the dedicated script: the web server will only send it after the execution of such a script anyway, and changing it out of context could mess with the transmission of the desired answer.
This is not a fixed rule: in cases where a standard answer is needed for example, it is possible to prepare it beforehand, and have the web server to use it for every answer, without repeating it every time.
The mechanic is:
- this function is used to set up a response string;
- after a dedicated script terminates, the web server sends the 'current' response string to the caller (that means the last prepared string, regardless when it was prepared).
Special care should only be used in asynchronous preparations of the response, if scripts unrelated to the OnWebserverCall event are involved.

Calling this function without parameter instead will enable the standard management of the packet in the web server (it's the way the invoked script can use to inform the web server that the API was not handled, and that standard work is still to be done).
Standard work includes management of file download requests, PDF viewer file requests, ESA API execution requests.
From this point of view, scripts programmed on the OnWebserverCall event are supposed to use this function to state whether they have already managed the received packet (interpreted as a recognized REST API), or not.

SENDRESTAPIREQUEST

Sends an HTTP(/s) REST API request to a server.

RESULT = **SENDRESTAPIREQUEST** (TRACE, TYPE, URL, PROXYADD, PROXYUSER, PROXYPWD, DATA, [HEADER1 [... [, HEADER8]]])

input

TRACE :	BOOL	a flag used to enable debug traces for the given command: TRUE means the exact command and response exchanged with the server will be traced in the runtime standard output
TYPE :	ANY_INT	states the type of request that is being sent; the supported codes are: **0** (RESTGET) **1** (RESTPOST) **2** (RESTPUT) **3** (RESTPATCH) **4** (RESTDELETE)

URL :	ANY_STRING	the complete URL that has to be accessed with the request;
		might have to include HTTP parameters, if required by the server protocol; for example, valid URLs could be:
		`http://the.server.com`
		`https://the.server.com/api`
		`https://the.server.com/api/v2?p1=1&p2=2`

PROXYADD :	ANY_STRING	IP address and port of the proxy; expected in the form "ip1.ip2.ip3.ip4:ipport"; can (and should) be an empty string if not needed
PROXYUSER :	ANY_STRING	user-name used for the authentication with the proxy; can be an empty string if the proxy is not used; ignored if PROXYADD is left empty; if needed, PROXYUSER and PROXYPWD must both be given
PROXYPWD :	ANY_STRING	password used for the authentication with the proxy; can be an empty string if the proxy is not used; ignored if PROXYADD is left empty; if needed, PROXYUSER and PROXYPWD must both be given
DATA :	ANY_STRING	the payload JSon string that might be required by the protocol in PUT, POST or PATCH requests; can be left empty if not needed

HEADER# :	ANY_STRING	[OPTIONAL] there are up to 8 parameters of this type:
		HEADER1, HEADER2, HEADER3, ..., HEADER8
		these parameters are used to specify header/meta information segments that have to be sent along the request;
		a typical usage case is the specification of the expected response content type, such as:
		`Accept: application/json`
		or the specification of authorization data, such as:
		`Authorization: Bearer b1094abc0-54a4-3eab-7213-877142c33fh3`
		should NOT be used instead to specify the content type of the request data (as in “Content-Type: application/json”), since the JSon format is the standard one expected by this function (when given through the DATA parameter), and is automatically declared by the system

output

RESULT :	UDINT	the numeric result code sent by the server in its response; the codes depend on the protocol implemented on the server, but HTTP standards usually apply, such as:

```
200   success
404   not found
500   internal error
```

and so on… (see HTTP specifications for details)

This function is used to send requests, as client, to a web server, in a HTTP/S REST API form.
It has a blocking behaviour, and won't return until the server sends its response (or until an error or a communication timeout has been recognized).

Along with the HTTP result numeric code directly given in the function return value, this function will store the (string) payload of the server response in the **RESTAPIRESPONSE** variable (often with JSon content).
Given the blocking behaviour of the function, this variable content will be immediately ready upon completion.
This variable will be updated by every **SENDRESTAPIREQUEST** call, regardless its actual return value; useful information might be included by the server even in error notifications, so finding it containing a valid JSon segment should not be used as proof that the request was successful.

Note that a function **GETRESTAPIRESPONSE** exists as well, made to return the exact same value currently held by the variable **RESTAPIRESPONSE**. The function is meant to simplify the access to the response string from JavaScript code implemented in the clients (a usage example can be found below).

Note: this function is currently implemented *only for Linux platforms*.

GETRESTAPIRESPONSE

Retrieves the string sent by a REST API server as response to the last request sent with a **SENDRESTAPIREQUEST**.

RESPONSE = **GETRESTAPIRESPONSE** ()

output

RESPONSE :	STRING	the data string received as response from the server

This function returns exactly the same value already stored in the variable **RESTAPIRESPONSE**.
It's meant to simplify the access to the response string from JavaScript code implemented in the clients (see below).

Note: this function is currently implemented *only for Linux platforms*.

JavaScript example

```
execSTScript('SENDRESTAPIREQUEST', ....function(err, val) {
    if (err)
        alert('SENDRESTAPIREQUEST: an error occured: ' + err);
    else if(val == 200) {
        execSTScript('GETRESTAPIRESPONSE', ....function(err, val) {
            let response = JSON.parse(val);
            ....
        }
        ....
```

SETTIMESSYTEM

Changes the time mode used by system outputs.

SETTIMESYSTEM (UTCMODE)

input		
UTCMODE :	BOOL	the time mode that has to be enabled; TRUE means UTC times will be used; FALSE means local times will be used

The system times mode affects the outputs of the runtime.
Being this function dedicated to the runtime server, the possible outputs are essentially the files exported by the different runtime modules.

GETTIMESYSTEM

Retrieves the time mode currently used by system outputs.

UTCMODE = **GETTIMESYSTEM** ()

output		
UTCMODE :	BOOL	the time mode currently enabled; TRUE means UTC times are being used; FALSE means local times are being used

The system times mode affects the outputs of the runtime.
Being this function dedicated to the runtime server, the possible outputs are essentially the files exported by the different runtime modules.

GETNUMCLIENTSWEB

Counts the number of clients currently connected through web server.

NUMBER = **GETNUMCLIENTSWEB** ()

output		
NUMBER :	UDINT	the number of clients currently connected (in web server mode)

Normally meant to count remote clients connected through generic browsers for reasons not necessarily related to the runtime itself.

GETNUMCLIENTSUI

Counts the number of clients currently connected through web socket.

NUMBER = **GETNUMCLIENTSUI** ()

output

NUMBER :	UDINT	the number of clients currently connected (in web socket mode)

Normally meant to count remote clients connected through generic browsers and running the runtime UI client provided by the runtime integrated web server.

GETNUMCLIENTSNET

Counts the number of clients currently connected through network project.

NUMBER = **GETNUMCLIENTSNET** ()

output		
NUMBER :	UDINT	the number of clients currently connected (in network project mode)

Normally meant to count remote machines connected as client of a network project.

LANGUAGEGET

Retrieves the ID of the language currently active in the server runtime.

LANGUAGE = **LANGUAGEGET** ()

output		
LANGUAGE :	UDINT	the current language identifier; this ID is defined as the (base-1) index of the language among those defined in the current project

LANGUAGESET

Activates a new language in the server runtime.

LANGUAGESET (LANGUAGE)

input

LANGUAGE :	UDINT	the new language identifier; this ID is defined as the (base-1) index of the language among those defined in the current project

LANGUAGENEXT

Activates the "next" language in the server runtime.

LANGUAGENEXT ()

The activated language is the one that comes after the current one in the list of the languages configured in the project.

LANGUAGEPREVIOUS

Activates the "previous" language in the server runtime.

LANGUAGEPREVIOUS ()

The activated language is the one that comes before the current one in the list of the languages configured in the project.

GETURL

Retrieves a file from an URL address.

GETURL (URL, FILE, PROXYADD, PROXYUSER, PROXYPWD, SERVERPORT, SERVERUSER, SERVERPWD)

input

URL :	ANY_STRING	address of file to be transferred; should be in form of an URL
FILE :	ANY_STRING	path and name local destination file
PROXYADD :	ANY_STRING	IP address and port of the proxy; expected in the form "ipaddress:ipport"; can be an empty string if not needed
PROXYUSER :	ANY_STRING	user-name used for the authentication with the proxy; can be an empty string if the proxy is not used
PROXYPWD :	ANY_STRING	password used for the authentication with the proxy; can be an empty string if the proxy is not used
SERVERPORT :	ANY_INT	server port
SERVERUSER :	ANY_STRING	user-name used for the authentication with the server; can be an empty string if the server is not used
SERVERPWD :	ANY_STRING	password used for the authentication with the server; can be an empty string if the server is not used

example

```
GETURL ( "http://198.168.100.1/image.jpg", "/home/esa/picture1.jpg",
         "proxy:8080", "PROXYUSER", "PROXYPASSWORD",
         1234, "admin", "admin" );

GETURL ( "http://198.168.100.2/image.jpg", "/home/esa/picture1.jpg",
         "198.168.99.1:8080", "PROXYUSER", "PROXYPASSWORD",
         0, "", "" );       // no server

GETURL ( "http://198.168.100.3/image.jpg", "/home/esa/picture1.jpg",
         "", "", "",        // no proxy or automatic management
         0, "", "" );       // no server
```

GETTICKS

Retrieves the current system ticks counter, in milliseconds.

TICKS = **GETTICKS** ()

output		
TICKS :	TIME	how long has the machine been operating; value in milliseconds resolution

The information is not related to the operativity time of the runtime: it's an information maintained by the operating system and basically tells "how long ago" the machine itself was switched on.
Can be used to quickly retrieve time markers when only relative measures are needed (this function is faster than those based on the actual system clock).

GETLTICKS

Retrieves the current system ticks counter, in nanoseconds.

TICKS = **GETLTICKS** ()

output		
TICKS :	LTIME	how long has the machine been operating; value in nanoseconds resolution

The information is not related to the operativity time of the runtime: it's an information maintained by the operating system and basically tells "how long ago" the machine itself was switched on.
Can be used to quickly retrieve time markers when only relative measures are needed (this function is faster than those based on the actual system clock).

SETRTC

Sets a new system date and time.

SETRTC (TIME)

input		
TIME :	ANY_DATE	new time to be set (local time mode); the value can be given in different date/time forms; the following types are supported: DATE : sets the new date and retains the current time TOD : sets the new time and retains the current date LTOD : sets the new time and retains the current date DT : sets both date and time LDT : sets both date and time

The function only affects the part of the current date/time that have actually been given by the provided parameter.

SETRTC_UTC

Sets a new system date and time.

SETRTC_UTC (TIME)

input		
TIME :	ANY_DATE	new date/time to be set (UTC time mode); the value can be given in different date/time forms; the following types are supported: DATE : sets the new date and retains the current time TOD : sets the new time and retains the current date LTOD : sets the new time and retains the current date DT : sets both date and time LDT : sets both date and time

The function only affects the part of the current date/time that have actually been given by the provided parameter.

GETRTCTOD

Retrieves the current system time.

TIME = **GETRTCTOD** ()

output		
TIME :	TOD	the current system time

The time in output is the current LOCAL time and is given in TOD form.

GETRTCLTOD

Retrieves the current system time.

TIME = **GETRTCLTOD** ()

output		
TIME :	LTOD	the current system time

The time in output is the current LOCAL time and is given in LTOD form.

GETRTCDATE

Retrieves the current system time.

DATE = **GETRTCDATE** ()

output		
DATE :	DATE	the current system date

The time in output is the current LOCAL time and is given in DATE form.

GETRTCDT

Retrieves the current system time.

TIME = **GETRTCDT** ()

output		
TIME :	DT	the current system date and time

The time in output is the current LOCAL time and is given in DT form.

GETRTCLDT

Retrieves the current system time.

TIME = **GETRTCLDT** ()

output		
TIME :	LDT	the current system date and time

The time in output is the current LOCAL time and is given in LDT form.

GETRTCTOD_UTC

Retrieves the current system time.

TIME = **GETRTCTOD_UTC** ()

output		
TIME :	TOD	the current system time

The time in output is the current UTC time and is given in TOD form.

GETRTCLTOD_UTC

Retrieves the current system time.

TIME = **GETRTCLTOD_UTC** ()

output		
TIME :	LTOD	the current system time

The time in output is the current UTC time and is given in LTOD form.

GETRTCDATE_UTC

Retrieves the current system time.

DATE = **GETRTCDATE_UTC** ()

output		
DATE :	DATE	the current system date

The time in output is the current UTC time and is given in DATE form.

GETRTCDT_UTC

Retrieves the current system time.

TIME = **GETRTCDT_UTC** ()

output		
TIME :	DT	the current system date and time

The time in output is the current UTC time and is given in DT form.

GETRTCLDT_UTC

Retrieves the current system time.

TIME = **GETRTCLDT_UTC** ()

output		
TIME :	LDT	the current system date and time

The time in output is the current UTC time and is given in LDT form.

REFRESHRTC

Align runtime clock to system RTC.

REFRESHRTC ()

The runtime clock (independent from the system RTC) is initialized at startup.
The information used by the clock are not limited to the date/time itself, but include knowledge of the current time-zone and the related offsets for standard and daylight-saving time.
Under normal conditions, the runtime re-aligns the time at regular (long) intervals, and never change the time-zone settings.
With this function it is possible to force an immediate refresh of all these time related information. This is useful in cases where external factors apply changes to the system time, such as time-zone management scripts, or remote NTP actions.

The alignment will also affect a set of system variables, used to give some information related to the current time zone and time offsets:
- **TIMEZONEOFFSET** : the offset, in seconds, between the current time-zone and the GMT;
- **TIMEOFFSETSTD** : the additional standard offset, between the local time and the UTC
- **TIMEOFFSETDST** : the additional daylight-saving time offset, between the local time and the UTC
- **TIMESTDSTART** : the transition date from daylight-saving time to standard time
- **TIMEDSTSTART** : the transition date from standard time to daylight-saving time

example

```
// LOCAL times should be identical to the UTC time plus the current time offsets

REFRESHRTC ();

_TRACE ( CSTRING ( GETRTCDT() ) );      // note that DT is numerically expressed in seconds:
_TRACE ( CSTRING ( GETRTCDT_UTC() ) );  // adding the offsets directly to the UTC will give the local time
_TRACE ( CSTRING ( GETRTCDT_UTC() + TIMEZONEOFFSET + TIMEOFFSETDST ) );

_TRACE ( CSTRING ( GETRTCTOD() ) );     // TOD instead is numerically expressed in milliseconds:
_TRACE ( CSTRING ( GETRTCTOD_UTC() ) ); // the offsets must be corrected to be added consistently
_TRACE ( CSTRING ( GETRTCTOD_UTC() + TIMEZONEOFFSET * 1000 + TIMEOFFSETDST * 1000 ) );
```

UTC_TO_LOCAL

Converts a date/time from UTC to local.

TIMELOCAL = **UTC_TO_LOCAL** (TIMEUTC)

input		
TIMEUTC :	ANY_DATE	the source (UTC) date/time can be given in different forms; the following types are supported: DT : the exact date/time will be converted LDT : the exact date/time will be converted TOD : the system applies calculations using the current date LTOD : the system applies calculations using the current date

output		
TIMELOCAL :	ANY_DATE	the converted (local) date/time; the output time replicates the input one

Note that when only a plain time is provided (if the date is not part of the given parameter type), then the system automatically assumes that the given time is intended for the current date.

LOCAL_TO_UTC

Converts a date/time from local to UTC.

TIMEUTC = **LOCAL_TO_UTC** (TIMELOCAL)

input

TIMELOCAL :	ANY_DATE	the source (local) date/time can be given in different forms; the following types are supported: DT : the exact date/time will be converted LDT : the exact date/time will be converted TOD : the system applies calculations using the current date LTOD : the system applies calculations using the current date

output

TIMEUTC :	ANY_DATE	the converted (UTC) date/time; the output time replicates the input one

Note that when only a plain time is provided (if the date is not part of the given parameter type), then the system automatically assumes that the given time is intended for the current date.

GETYEAR

Retrieves the "year" component of a given date.

YEAR = **GETYEAR** (DATE)

input		
DATE :	ANY_DATE	date from which the year has to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DATE, DT, LDT (only date/times that actually contain years are acceptable)
output		
YEAR :	UINT	the year component of the given date/time

GETMONTH

Retrieves the "month" component of a given date.

MONTH = **GETMONTH** (DATE)

input		
DATE :	ANY_DATE	date from which the month has to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DATE, DT, LDT (only date/times that actually contain months are acceptable)

output		
MONTH :	UINT	the month component of the given date/time

GETDAY

Retrieves the "day" component of a given date.

DAY = **GETDAY** (DATE)

input

DATE :	ANY_DATE	date from which the day has to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DATE, DT, LDT (only date/times that actually contain days are acceptable)

output

DAY :	UINT	the day component of the given date/time

GETWEEKDAY

Finds the "day of weak" of a given date.

WEEKDAY = **GETWEEKDAY** (DATE)

input		
DATE :	ANY_DATE	date from which the day of week has to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DATE, DT, LDT (only date/times that actually contain dates are acceptable)

output		
WEEKDAY :	UINT	the day of week of the given date/time; this is given as a value in the range 0..6 (0=Sunday, 1=Monday, …, 6=Saturday)

GETHOURS

Retrieves the "hours" component of a given time.

HOURS = **GETHOURS** (TIME)

input		
TIME :	ANY_DATE	time from which the hours have to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DT, LDT, TOD, LTOD (only date/times that actually contain hours are acceptable)

output		
HOURS :	UINT	the hours component of the given date/time

GETMINUTES

Retrieves the "minutes" component of a given time.

MINUTES = **GETMINUTES** (TIME)

input		
TIME :	ANY_DATE	time from which the minutes have to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DT, LDT, TOD, LTOD (only date/times that actually contain minutes are acceptable)
output		
MINUTES :	UINT	the minutes component of the given date/time

GETSECONDS

Retrieves the "seconds" component of a given time.

SECONDS = **GETSECONDS** (TIME)

input		
TIME :	ANY_DATE	time from which the seconds have to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DT, LDT, TOD, LTOD (only date/times that actually contain seconds are acceptable)
output		
SECONDS :	UINT	the seconds component of the given date/time

GETMSECONDS

Retrieves the "milliseconds" component of a given time.

MSECONDS = **GETMSECONDS** (TIME)

input		
TIME :	ANY_DATE	time from which the milliseconds have to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DT, LDT, TOD, LTOD (only date/times that actually contain "times" are acceptable); note that in case of DT, the result is always 0, since the DT resolution is in seconds
output		
MSECONDS :	UINT	the milliseconds component of the given date/time (this is the decimal part of the seconds expressed in milliseconds)

GETNSECONDS

Retrieves the "nanoseconds" component of a given time.

NSECONDS = **GETNSECONDS** (TIME)

input		
TIME :	ANY_DATE	time from which the nanoseconds have to be extracted; despite the 'ANY_DATE' declaration, this parameter can only have one of the following types: DT, LDT, TOD, LTOD (only date/times that actually contain "times" are acceptable); note that in case of DT, the result is always 0, since the DT resolution is in seconds

output		
NSECONDS :	UDINT	the nanoseconds component of the given date/time (this is the decimal part of the seconds expressed in nanoseconds)

MAKETOD

Creates a time value using the given components.

TIME = **MAKETOD** (HOURS, MINUTES, SECONDS, MILLISECONDS)

input

HOURS :	ANY_INT	hours component
MINUTES :	ANY_INT	minutes component
SECONDS :	ANY_INT	seconds component
MILLISECONDS :	ANY_INT	milliseconds component

output

TIME :	TOD	the time of day (TOD) resulting from the provided components

MAKELTOD

Creates a time value using the given components.

TIME = **MAKELTOD** (HOURS, MINUTES, SECONDS, NANOSECONDS)

input		
HOURS :	ANY_INT	hours component
MINUTES :	ANY_INT	minutes component
SECONDS :	ANY_INT	seconds component
NANOSECONDS :	ANY_INT	nanoseconds component

output		
TIME :	LTOD	the time of day (LTOD) resulting from the provided components

MAKEDATE

Creates a date value using the given components.

DATE = **MAKEDATE** (YEAR, MONTH, DAY)

input		
YEAR :	ANY_INT	year component
MONTH :	ANY_INT	month component
DAY :	ANY_INT	day component

output		
DATE :	DATE	the date resulting from the provided components

MAKEDT

Creates a date/time value using the given components.

TIME = **MAKEDT** (YEAR, MONTH, DAY, HOURS, MINUTES, SECONDS)

input		
YEAR :	ANY_INT	year component
MONTH :	ANY_INT	month component
DAY :	ANY_INT	day component
HOURS :	ANY_INT	hours component
MINUTES :	ANY_INT	minutes component
SECONDS :	ANY_INT	seconds component

output		
TIME :	DT	the date/time (DT) resulting from the provided components

MAKELDT

Creates a date/time value using the given components.

TIME = **MAKELDT** (YEAR, MONTH, DAY, HOURS, MINUTES, SECONDS, NANOSECONDS)

input		
YEAR :	ANY_INT	year component
MONTH :	ANY_INT	month component
DAY :	ANY_INT	day component
HOURS :	ANY_INT	hours component
MINUTES :	ANY_INT	minutes component
SECONDS :	ANY_INT	seconds component
NANOSECONDS :	ANY_INT	nanoseconds component

output		
TIME :	LDT	the date/time (LDT) resulting from the provided components

GETCLIENTID

Retrieves the ID of the calling client.

ID = **GETCLIENTID** ()

output		
ID :	UDINT	the ID (positional index of client in connections list) of the client responsible of the current script execution

If the script is being executed due to a server call, and no client is involved, then the function returns 0xFFFFFFFF.
Otherwise (a client is involved) one of the following applies.

If matrix-users are in use, then:
- if the client is among those configured in the geographic authorizations' matrix, then the returned ID is the position of the client in the configured clients list (base-0);
- if the client is a guest, then the ID is assigned at the moment of the connection; the given ID corresponds to the first free slot, higher than the reserved ones, and within a preconfigured maximum limit.
If level-users are in use, then the behaviour is that of the guest clients above: all clients receive their ID at the moment of their connection, chosen between 0 and a configured maximum limit.

If too many clients are connected (more than the configured maximum number), then all the "extra" clients will receive the same ID, equal to the highest possible slot.

SETCLIENTDEBUGRIGHTS

Chooses the debugging rights of a given client.

SETCLIENTDEBUGRIGHTS (ID, RIGHTS)

input		
ID :	UDINT	ID of the interested client; this parameter is used only in case of "matrix" users; ignored instead in case of "level" users; can be used to specify special selection cases: **SYSALLCLIENTS** (0xFFFFFFFF) can be used to indicate that the selection has to be applied to all the clients contemporarily (in case of level-users this is the systematic implicit working mode); **SYSCALLINGCLIENT** (0xFFFFFFFE) can be used to indicate that the selection has to be applied to the client that requested the execution of the function (possible only if this client actually exists, not if the function execution originated from the server);
RIGHTS :	BOOL	the rights selection; TRUE means the given client(s) is authorized to run debugging sessions in its browser (FALSE if not)

Used to enable or disable debugging permissions for one or more clients.

In case of "level" users, there is no ability to identify (at design time) specific clients, so this operation is always considered global (the debugging authorization is always meant for all the clients or none of them).
In this case, the 1st function parameter (ID) is ignored.

In case of "matrix" users, clients IDs are configured and known at design time, so dedicated logic can be programmed.
In this case the 1st function parameter (ID) is meaningful, and can be used to select specific clients. Programmers are allowed to choose a single client (with given ID), or to extend the selection to all the clients (**SYSALLCLIENTS**); also if the script execution originated from a client event, then programmers are allowed to apply the selection to the calling client itself (**SYSCALLINGCLIENT**).
If a single client has been selected, and it turned out to be a "guest", then the selection is extended to all the possible guests (similarly to level-users behaviour, in case of matrix-users guests, the authorization can only be given to all the guests or no guest at all).

GETCLIENTDEBUGRIGHTS

Retrieves the current debugging rights of a given client.

RIGHTS = **GETCLIENTDEBUGRIGHTS** (ID)

input		
ID :	UDINT	ID of the interested client; only meaningful in case of "matrix" users; ignored in case of "level" users; SYSALLCLIENTS and SYSCALLINGCLIENT special cases can be used; see SETCLIENTDEBUGRIGHTS for extensive details
output		
RIGHTS :	BOOL	the rights selection; TRUE means the given client(s) is authorized to run debugging sessions in its browser (FALSE if not)

Allows to see whether given clients have been granted or rejected the right to execute debugging sessions within their browser.

The same basic principles described in the SETCLIENTDEBUGRIGHTS about the supported ID parameter values, and related behaviours and mechanics, are still valid.
Specifically, in case of level-users the authorization retrieved is always a "global" one (currently valid for all the clients), while in case of matrix-users the authorization could be client-specific. In this last case, keep in mind that with matrix-users:
- if a "guest" client is targeted, then the authorization retrieved is always the same as that of any other guest;
- if a "global" authorization is asked (SYSALLCLIENTS), then a meaningful value is obtained only if a global authorization has really been explicitly assigned with a previous call to SETCLIENTDEBUGRIGHTS in SYSALLCLIENTS mode.

GETCLIENTDEBUGSTATE

Finds out whether a given client has ever executed a (legitimate or not) debugging session.

STATE = **GETCLIENTDEBUGSTATE** (ID)

input		
ID :	UDINT	ID of the interested client; only meaningful in case of "matrix" users; ignored in case of "level" users; SYSALLCLIENTS and SYSCALLINGCLIENT special cases can be used; see SETCLIENTDEBUGRIGHTS for extensive details
output		
STATE :	UDINT	a code describing the debug occurrences for the given client; expected values are: **0** (SYSDEBUGNONE) : nothing (debug-related) ever happened; **1** (SYSDEBUGREJECTED) : at least a debug session has been denied; **2** (SYSDEBUGACCEPTED) : at least a debug session has been accepted; →NOTE: up to clients' implementation: this last case might be impossible if clients are implemented to only provide "rejected" events, and not "accepted" ones **3** (SYSDEBUGREJECTED | SYSDEBUGREJECTED) : (the two events above combined in 'or'); both the events occurred at least once in the client;

Global requests (implicit with level-users, or explicitly stated in SYSALLCLIENTS mode with matrix-users) are used to find out whether any of the existing clients has ever tried to debug the application. Targeting specific clients is possible only in case of matrix-users; limits and behaviours are always those described for SETCLIENTDEBUGRIGHTS and GETCLIENTDEBUGRIGHTS.

Note that the retrieved information doesn't necessarily report events related to the current instances of the clients, but simply states that have been recorded from clients' notifications at any time in the current execution session of the server: different clients might have disconnected and reconnected at runtime, but the registrations returned by this call still remain.
The reset of the recorded states is only possible by writing the system tags "SYS_ClientDebugState" and "SYS_AnyClientDebugState" (affecting the 'rejected' conditions).

SETCLIENTOFFSCAN

Sets the offscan state of a given client.

SETCLIENTOFFSCAN (ID, STATE)

input		
ID :	UDINT	ID of the interested client; usage identical to **SETCLIENTDEBUGRIGHTS**: this parameter is used only in case of "matrix" users; ignored instead in case of "level" users; can be used to specify special selection cases: **SYSALLCLIENTS** (0xFFFFFFFF) can be used to indicate that the selection has to be applied to all the clients contemporarily (in case of level-users this is the systematic implicit working mode); **SYSCALLINGCLIENT** (0xFFFFFFFE) can be used to indicate that the selection has to be applied to the client that requested the execution of the function (possible only if this client actually exists, not if the function execution originated from the server);
STATE :	BOOL	the offscan state selection; TRUE means the given client(s) is allowed to communicate with the server; FALSE if not allowed: in this case the server communication will be cut

Used to enable or disable the communication between the server and one or more clients.
The communication is cut at WebSocket level.
The cut might affect both local and remote clients, and servers sharing tags in a network project.

Just like the mentioned **SETCLIENTDEBUGRIGHTS**, in case of "level" users, there is no ability to identify (at design time) specific clients, so this operation is always considered global (the offscan state is always meant for all the clients).
In this case, the 1st function parameter (ID) is ignored.

In case of "matrix" users, clients IDs are configured and known at design time, so dedicated logic can be programmed.
In this case the 1st function parameter (ID) is meaningful, and can be used to select specific clients. Programmers are allowed to choose a single client (with given ID), or to extend the selection to all the clients (**SYSALLCLIENTS**); also if the script execution originated from a client event, then programmers are allowed to apply the selection to the calling client itself (**SYSCALLINGCLIENT**).
If a single client has been selected, and it turned out to be a "guest", then the selection is extended to all the possible guests (similarly to level-users behaviour, in case of matrix-users guests, the state can only be set for all the guests as a whole).

GETCLIENTOFFSCAN

Retrieves the current offscan state of a given client.

STATE = **GETCLIENTOFFSCAN** (ID)

input		
ID :	UDINT	ID of the interested client; only meaningful in case of "matrix" users; ignored in case of "level" users; SYSALLCLIENTS and SYSCALLINGCLIENT special cases can be used; see SETCLIENTOFFSCAN for extensive details

output		
STATE :	BOOL	the current offscan state; TRUE means the given client(s) is currently in offscan (FALSE if not)

Allows to see whether given clients are currently operating in offscan mode (with their communication with the server cut or limited).

The same basic principles described in the SETCLIENTOFFSCAN about the supported ID parameter values, and related behaviours and mechanics, are still valid.
Specifically, in case of level-users the state retrieved is always a "global" one (currently valid for all the clients), while in case of matrix-users the state could be client-specific. In this last case, keep in mind that with matrix-users:
- if a "guest" client is targeted, then the state retrieved is always common to all the guests;
- if a "global" state is asked (SYSALLCLIENTS), then a meaningful response is obtained only if a global state has really been explicitly set with a previous call to SETCLIENTOFFSCAN in SYSALLCLIENTS mode.

SETCLIENTKEYBURST

Enables or disables the clients ability to collect streams of keys sent in burst by physical keyboards.
The functionality is meant to manage codes sent by barcode readers operating in keyboard emulation mode.

SETCLIENTKEYBURST (ID, STATE)

input		
ID :	UDINT	ID of the interested client; usage rules are similar to the functions above, but with different special cases: - specific clients IDs are only supported in case of "matrix" users; if provided in case of "level" users, the IDs are simply ignored and the function behaves as if invoked for a SYSALLCLIENTS case (see below); - the following codes can be used to specify special selection cases (these codes are always acceptable, either in "matrix" or "level" mode): SYSALLCLIENTS (0xFFFFFFFF) can be used to indicate that the selection has to be applied to all the clients contemporarily; SYSCALLINGCLIENT (0xFFFFFFFE) can be used to indicate that the selection has to be applied to the client that requested the execution of the function (possible only if this client actually exists, not if the function execution originated from the server); SYSLOCALCLIENTS (0xFFFFFFFD) can be used to indicate that the selection has to be applied to all the clients currently connected to the sever in localhost (usually one only, but multiple local clients are supported as well);
STATE :	BOOL	the key-bursts management state selection; TRUE means the given client(s) is allowed/required to collect key-bursts; FALSE otherwise

Used to enable or disable the key-bursts collection activity of targeted clients.
The clients with this feature enabled are supposed to monitor keys pressure events, identify those received in burst (at least a minimum number of keys with minimal delay in between), and store the collected streams in dedicated system variables.

As mentioned above, special selection cases are always supported:
SYSALLCLIENTS always extends the selection to all the clients (whether they are already connected or not),
SYSCALLINGCLIENT always targets the client that invoked the script execution (provided such client exists),
SYSLOCALCLIENTS always extends the selection to all the clients currently connected in localhost (only those already connected, not those that might connect in the future).
If a specific client ID is given instead, the selection is only allowed in case of matrix-users. In case of level-users, there is no ability to identify specific clients at design time, so this kind of selection is always changed into a global (SYSALLCLIENTS) request.
Also, if a specific client has been selected (matrix-users mode), and it turned out to be a "guest", then the selection is extended to all the possible guests (similarly to level-users behaviour, in case of matrix-users guests there is no possibility to pre-identify specific clients, so the state can only be set for all the guests as a whole).

GETCLIENTKEYBURST

Retrieves the current key-burst management state of a given client.
The state is initialized at FALSE for all clients, and can be changed with SETCLIENTKEYBURST calls.

STATE = **GETCLIENTKEYBURST** (ID)

input		
ID :	UDINT	ID of the interested client; can be the ID of a specific client (supported only in case of "matrix" users), or the code of a special selection (supported special cases are SYSALLCLIENTS, SYSCALLINGCLIENT and SYSLOCALCLIENTS); see SETCLIENTKEYBURST for extensive details
output		
STATE :	BOOL	the current key-burst management state; TRUE means the management is currently enabled in the given client(s); FALSE if not

Allows to see whether given clients are currently managing key-bursts or not.

The same basic principles described in the SETCLIENTKEYBURST about the supported ID parameter values, and related behaviours and mechanics, are still valid.

SAVESCREEN

Save in a designated file an image of whatever is currently displayed on the server screen.

SAVESCREEN (FILENAME)

input		
FILENAME :	ANY_STRING	name of the destination file; the format of the created file is implicitly taken from the extension specified in the given name; supported extensions/formats are: BMP, PNG, JPG

BEEP

Sounds a buzzer beep.
Executed by the runtime on the server machine.

BEEP (DURATION, FREQUENCY, SYNC)

input

DURATION :	ANY_INT	the duration of the beep; can be given as a plain numeric value, expressed in milliseconds, or with a native duration type, such as a TIME or LTIME value
FREQUENCY :	ANY_INT	the frequency of the sound, in Hertz; depending on the HMI hardware in use, this parameter might be ignored (some embedded panels buzzers are limited to a fixed frequency);
SYNC :	BOOL	TRUE if the script must be kept blocked for the whole duration of the sound; FALSE if the script execution can proceed even while the sound is active

BEEPON

Switches on a buzzer sound.
This functionality is limited to embedded platforms, and is executed on the server machine.

BEEPON (FREQUENCY)

input

FREQUENCY :	ANY_INT	[OPTIONAL]	the frequency of the sound, in Hertz; depending on the HMI hardware in use, this parameter might be ignored (some panels buzzers have a fixed frequency)

The sound will remain active until a subsequent **BEEPOFF** or **BEEP** command is given.
This command is used in cases where the script itself is meant to handle the duration of the sound.

BEEPOFF

Switches off a buzzer sound.
This functionality is limited to embedded platforms, and is executed in the server machine.

BEEPOFF ()

Expected to be used after a **BEEPON** command, in cases where the script itself is meant to handle the duration of the sound.

LIGHTUP

Increases the brightness of the display.
This functionality is limited to embedded platforms, and is executed in the server machine.

LIGHTUP ()

LIGHTDOWN

Decreases the brightness of the display.
This functionality is limited to embedded platforms, and is executed in the server machine.

LIGHTDOWN ()

LIGHTSET

Sets the brightness of the display to a given level.
This functionality is limited to embedded platforms, and is executed in the server machine.

LIGHTSET (BRIGHTNESS)

input		
BRIGHTNESS :	ANY_NUM	the needed brightness level; can be a precise level (integer in the range implemented by the hardware), or a percentage (floating point) of the supported maximum

Since different HMI hardware support different ranges of levels, this function allows a management of all the panels in a homogeneous way:
- if the BRIGHTNESS parameter is given as an integer value (one of the ANY_INT family type), then it is treated as a precise level value; programmers are supposed to know the exact range supported by the panel in use (see **LIGHTGETMAX**);
- if the BRIGHTNESS parameter is given as a floating point value (one of the ANY_REAL family type), then its value is supposed to be in the range 0÷100 and is treated as a percentage of the range known by the runtime

LIGHTGET

Retrieves the current level value of the brightness of the display.

BRIGHTNESS = **LIGHTGET** ()

output		
BRIGHTNESS :	LREAL	the current brightness level; the returned value is in the range 0÷100 and shows the current level as a percentage, between a 'switched off' state (0) and the maximum brightness supported by the hardware (100)

Note that the functions **LIGHTSET**, **LIGHTUP** and **LIGHTDOWN** (can) work with absolute brightness levels, where each step can be bigger or smaller than a percentage point.
This means that is possible and normal to face situations in which a level is set as an absolute value, then an UP/DOWN in requested, and finally, when read, the percentage difference doesn't match the number of UP/DOWN steps.

LIGHTGETMAX

Retrieves the maximum level value of the brightness of the display supported by the hardware.

MAXIMUM = **LIGHTGETMAX** ()

output

MAXIMUM :	UDINT	the maximum brightness level; the returned value is the absolute limit of the range implemented by the hardware light dimming functionality

Different embedded HMI hardware will return different limits.
PCs don't support the dimming functionality, and will always return 0.

example

```
VAR
   absolute : UDINT;
   percentage : REAL;
END_VAR;

absolute := LIGHTGETMAX ();              // a 7" imx6 panel will return 32

percentage := 40.0;
LIGHTSET (percentage);                   // the same panel is now set to 12 (40/100*32 = 12.8)
percentage := LIGHTGET ();               // the same panel will return 37.5 (12/32*100)

absolute := 16;
LIGHTSET (absolute);
percentage := LIGHTGET ();               // the same panel will return 50.0 (16/32*100)

LIGHTUP ();
percentage := LIGHTGET ();               // the same panel will return 53.125 (17/32 = 50+100/32)
```

SHOWTASKBAR

Shows or hides the system desktop TaskBar.

SHOWTASKBAR (SHOW)

output		
SHOW :	BOOL	TRUE to show the TaskBar FALSE to hide the TaskBar

This function is only effective in Windows (with standard Explorer desktop).

SETHHLEDSTATE

Switches on/off the led of a Handheld panel.

SETHHLEDSTATE (LED, STATE)

output		
LED :	UDINT	ID of the interested LED (IDs are defined in the project's compiled leds configuration)
STATE:	BOOL	New state for the given led; TRUE to switch it ON, FALSE to switch it OFF

This function is only effective on Handheld panels, and only in case of projects that explicitly enable the leds management.
It is ignored in case of different platforms (Windows) or panels.

< VARIABLES >

The following are the variables usable to share information and directives for the system management:

Runtime identification:

RT_PLATFORM
type UDINT
access R
gives a numeric code usable to identify the type of hardware platform where the runtime is running;
information is stored in groups of 4 bits each:
the 1st digit (less significant 4 bits) can be:
0x0**0** (SYSARM) : the machine is an **embedded** ARM-based hardware
0x0**1** (SYSPC) : the machine is a **PC**
0x0**F** (SYSUNK) : the machine hardware is unknown
the 2nd digit can be:
0x0**0** (SYSWINDOWS) : the platform is based on a **Windows** operating system
0x0**1** (SYSLINUX) : the platform is based on a **Linux** operating system
0x0**F** (SYSUNK) : the operating system is unknown

RT_WORKMODE
type UDINT
access R
gives a numeric code usable to identify the current working mode of the runtime;
the expected possible values are:
0 (SYSFULLSYSTEM) : both client and server modules are loaded; the loaded client will provide the project pages window
2 (SYSSERVERONLY) : only the server modules are loaded; the server runs in background or console; no window is created; supposed to be used by clients implementing their own window
3 (SYSSTANDALONE) : only the server modules are loaded; a basic window is provided to keep the server alive as stand-alone application
In the current systems configuration, only the SYSSERVERONLY mode should happen

RT_SIMULATION
type BOOL
access R
states whether the runtime is currently running in (offline) simulation mode (TRUE) or not (FALSE);
the simulation flag only states the ability of the runtime to work online with external devices; no other functionality is limited

RT_VERSION
type STRING
access R
gives a version code, in string form, of the current implementation of the server runtime;
structured in 3 levels, the format is "major.minor.progress"
starting from “1.0.1”; the version progress should follow the runtime development itself

RT_VERSION_MAJOR
type UINT
access R
gives the numeric value of the 'major' level of the runtime version;
see RT_VERSION above

RT_VERSION_MINOR
type UINT

access R
gives the numeric value of the 'minor' level of the runtime version;
see **RT_VERSION** above

RT_VERSION_PROGRESS
type UINT
access R
gives the numeric value of the 'progress' level of the runtime version;
see **RT_VERSION** above

ST_VERSION
type STRING
access R
gives a version code, in string form, of the current implementation of the ST scripting engine;
normally set to the same version as the present document (see this document history chapter for evolution details)

RT_SESSION
type UDINT
access R
gives a numeric identifier of the current execution session of the server runtime; this code is randomly generated by the runtime every time it is executed;
it's the code that is normally used by clients to identify server sessions after communication losses

Errors management:

ERRNO

type ULINT
access R/W

gives the code of the last potentially blocking error encountered by the scripts;
by default, blocking errors halt the execution of a script; in several cases though they can be intercepted and handled:
if the option ST_OPTION HANDLE_ERRORS is set, then the errors won't result in scripts terminations; their code will be stored in this variable instead;

to handle the errors:
- ST_OPTION HANDLE_ERRORS allows to handle errors through this variable;
- ST_OPTION BLOCKING_ERRORS resets the default blocking behaviour;

to reset the error code:
- use the function ERRORRESET (will reset ERRMSG as well);
- manually clear the variable (ERRNO := 0);

to inspect the meaning of error code:
- use ERRMSG to retrieve a description text of the error in ERRNO;
- use ERRORGETMESSAGE to retrieve a description of any error code;
- use ERRORGETMODULE to retrieve the name of the involved runtime module

errors that can be handled include:
- assignments to unreachable variables
- errors from evaluations of expressions in assignments
- faults executing embedded key functions
- faults executing user defined functions

there are errors though that will never be trappable; examples are:
- any logic or syntactic error identified at validation time
- errors from evaluation of expressions used as controls in decisional constructs

ERRMSG

type WSTRING
access R/W

gives a textual description of the error currently stored in ERRNO (see above for details);
can be cleared manually (ERRMSG := '') or calling the function ERRORRESET (will reset ERRNO as well)

FXRESULT

type ULINT
access R/W

gives the result of the execution of the last called embedded key function (meaning all the functions described in this document);
0 means the last function execution was successful;
any other value identifies an error instead;
its value can be manually reset (FXRESULT := 0);
as already stated, to inspect the error codes:
- use ERRORGETMESSAGE to retrieve a description text;
- use ERRORGETMODULE to retrieve the name of the involved runtime module

Clock:

TIMEZONEOFFSET

type DINT
access R
the base offset, in seconds, between the current time-zone time, and the UTC time;
the value is initialized at startup and then changed only upon explicit calls of the **REFRESHRTC** function

TIMEOFFSETSTD

type DINT
access R
an additional offset, in seconds, applied to the current local time during standard time days (see **TIMEOFFSETDST** too);
for example, during standard time days, the current local time is equal to the current UTC time + **TIMEZONEOFFSET** + **TIMEOFFSETSTD**;
the value is initialized at startup and then changed only upon explicit calls of the **REFRESHRTC** function

TIMEOFFSETDST

type DINT
access R
an additional offset, in seconds, applied to the current local time during daylight-saving days (see **TIMEOFFSETSTD** too);
for example, during daylight-saving time days, the current local time is equal to the current UTC time + **TIMEZONEOFFSET** + **TIMEOFFSETDST**;
the value is initialized at startup and then changed only upon explicit calls of the **REFRESHRTC** function

TIMESTDSTART

type DT
access R
the starting date of the 'standard' time for the current year (transition date from daylight-saving time to standard time); see **TIMEDSTSTART** too;
the value is initialized at startup and then changed only upon explicit calls of the **REFRESHRTC** function

TIMEDSTSTART

type DT
access R
the starting date of the 'daylight-saving' time for the current year (transition date from standard time to daylight-saving time); see **TIMESTDSTART** too;
the value is initialized at startup and then changed only upon explicit calls of the **REFRESHRTC** function

System:

RESTAPIRESPONSE

type STRING
access R
used to store the data string received from a server as response to a **SENDRESTAPIREQUEST** call;
updated at every call, not only upon success

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods, or checking values from variables or methods results:

symbolic	value	relevant methods
SYSRUNASYNCUNIQUE	0	RUNAPPLICATION
SYSRUNASYNCFREE	1	RUNAPPLICATION
SYSRUNBLOCKING	2	RUNAPPLICATION
SYSBLANKS	0xFFFFFFFF	SPLIT, TRIM, LTRIM, RTRIM
SYSALLCLIENTS	0xFFFFFFFF	GET/SETCLIENTDEBUGRIGHTS/STATE, GET/SETCLIENTOFFSCAN, GET/SETCLIENTKEYBURST
SYSCALLINGCLIENT	0xFFFFFFFE	GET/SETCLIENTDEBUGRIGHTS/STATE, GET/SETCLIENTOFFSCAN, GET/SETCLIENTKEYBURST
SYSLOCALCLIENTS	0xFFFFFFFD	GET/SETCLIENTKEYBURST
SYSDEBUGNONE	0	GETCLIENTDEBUGSTATE
SYSDEBUGREJECTED	1	GETCLIENTDEBUGSTATE
SYSDEBUGACCEPTED	2	GETCLIENTDEBUGSTATE
SYSUNK	0x0F	RT_PLATFORM
SYSARM	0	RT_PLATFORM
SYSPC	1	RT_PLATFORM
SYSWINDOWS	0	RT_PLATFORM
SYSLINUX	1	RT_PLATFORM
SYSFULLSYSTEM	0	RT_WORKMODE
SYSSERVERONLY	2	RT_WORKMODE
SYSSTANDALONE	3	RT_WORKMODE
RESTGET	0	SENDRESTAPIREQUEST
RESTPOST	1	SENDRESTAPIREQUEST
RESTPUT	2	SENDRESTAPIREQUEST
RESTPATCH	3	SENDRESTAPIREQUEST
RESTDELETE	4	SENDRESTAPIREQUEST

5. COMMON - FILES

Files paths conventions

An initial note regarding the system behaviour with regards to the interpretation of the files paths and names, shared by all the files related implemented functions.

The paths resolution conventions are as follows:
- if a file path+name starts with "****" or "**/**" or (on windows) with a unit specification (e.g. "c:"), then it's considered an **absolute path**, and it's kept as it is; this model can be used to target anything on a given machine;
- if a file path+name starts with "**$**" and the symbolic name of a known **conventional system folder** (e.g. "$RESOURCES"), then everything following it is considered relative to the actual folder as declared in the ESA.INI file (see the list below); this model can be used to target resources located in specific system folders, without the need to know the exact location of the folder itself;
- if a file path+name is NOT absolute (as defined above) and does NOT start with a known system folder identifier, then it is considered **relative to the "LOG" folder** declared in the ESA.INI file; this model can be used to work with files in a folder and subfolders known to be freely accessible on the machine (also note that the "LOG" folder can be explicitly given as system conventional folder with the symbolic "$LOG").

Regarding the conventional system folders, the following symbolics are supported by the runtime:

`$RESOURCES`	
`$LOG`	
`$CONFIG`	
`$IMAGES`	
`$FONTS`	
`$HTML`	
`$USB`	(the path of the 1st detected USB device; in case there is no USB device, redirects on $LOG; used in cases where a path must be resolved somehow, even if the desired device is not connected)
`$USB#`	(the path of the #th detected USB device; if the device doesn't exist, the path remains empty; used in cases where the exact path of a specific device is required; usable to detect the existence of a device)

They must be given in uppercase and must be placed at the beginning of a file path+name string.
Everything not perfectly matching the listed symbolics and the rules above, will not be recognized as a system folder identifier, and will most likely be treated as a plain piece of string of a relative path.

Finally note that in specific deployments, or from some point onwards, some of the models above might be cut off and rendered unusable, in order to limit the accessibility of specific (or all) machines.

FILE_EXIST

Checks whether a file with given name and path exists.
Can be used to detect both files and directories.

EXIST = **FILE_EXIST** (NAME)

input		
NAME :	ANY_STRING	path and name of the target file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output		
EXIST :	BOOL	TRUE if the file exists FALSE if it doesn't

FILE_COPY

Copies a file.

FILE_COPY (SOURCE, DESTINATION, OVERWRITE)

input		
SOURCE :	ANY_STRING	path and name of the source file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
DESTINATION :	ANY_STRING	path and name of the destination file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
OVERWRITE :	BOOL	TRUE if an already existing destination can be overwritten; FALSE if overwriting is forbidden

The destination folder must exist.
Both the source and the destination elements must be accessible and free.
If the destination already exists, the OVERWRITE parameter states whether it can be overwritten or not; if not, the function fails.

FILE_DELETE

Removes a file.

FILE_DELETE (NAME)

input

NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

FILE_RENAME

Renames or moves a file.

FILE_RENAME (SOURCE, DESTINATION)

input		
SOURCE :	ANY_STRING	path and name of the source file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
DESTINATION :	ANY_STRING	path and name of the destination file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

When the names are given, the use of absolute paths is recommended, since different file systems use different policies with respect to the concept of current/relative path. When relative paths are used, the programmer is expected to know the behaviour of the file system in use.

The function can be used to actually move a file, depending on the given DESTINATION parameter.
If the same path is given for source and destination, then the function acts as a plain rename; if different paths are given, then the file is moved from the source to the destination one.
If only names are given, then the system assumes the 'current' path has to be used, and inconsistencies might arise between different platforms and file systems.

FILE_CREATEDIR

Creates a new directory.

FILE_CREATEDIR (NAME)

input

NAME :	ANY_STRING	path and name of the directory; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

FILE_DELETEDIR

Removes a directory.
A recursive operation can be requested.

FILE_DELETEDIR (NAME, RECURSIVE)

input

NAME :	ANY_STRING	path and name of the directory; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
RECURSIVE :	BOOL	TRUE if even the content of the directory has to be (recursively) removed

If no recursivity is enforced, then the given directory is removed only if it's already empty; the operation fails otherwise.
If recursivity is required instead, then even the content of the directory is removed; recursively, even subdirectories are treated accordingly.

FILE_GETSIZE

Retrieves the size of a file.
The file must not be currently opened by anyone, scripts included.

SIZE = **FILE_GETSIZE** (NAME)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output		
SIZE :	ULINT	size in bytes of the specified file

Not to be confused with the **FILE_GETLENGTH**, used on opened streams.

FILE_SETSIZE

Sets a new size for a file.

FILE_SETSIZE (NAME, SIZE)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
SIZE :	ANY_UNSIGNED	new file size

If the file is currently bigger than the required size, then it is truncated.
If it is currently smaller, then it is filled with 0s bytes up to the new size.
If the file doesn't exist at all, then a new one is created and filled with the proper amount of 0s.

FILE_GETTIMECREATION

Retrieves the creation date and time of a given file.

TIME = **FILE_GETTIMECREATION** (NAME)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output		
TIME :	LDT	the file creation date and time

FILE_GETTIMEWRITE

Retrieves the last write date and time of a given file.

TIME = **FILE_GETTIMEWRITE** (NAME)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output		
TIME :	LDT	the file last write's date and time

FILE_GETTIMEACCESS

Retrieves the last access date and time of a given file.

TIME = **FILE_GETTIMEACCESS** (NAME)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output		
TIME :	LDT	the file last access date and time

FILE_ISDIRECTORY

Checks whether a given name is identifying a file or a directory.

DIR = **FILE_ISDIRECTORY** (NAME)

input

NAME :	ANY_STRING	path and name of the file or directory; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output

DIR :	BOOL	TRUE if the parameter identifies a directory FALSE if it doesn't identify a directory; note that if 'special' files (such as fifo, blocks, devices, sockets, and all the different elements that Linux adds to the file system) are forcibly given, they would fall under the 'FALSE' category

FILE_FINDFIRST

Starts a files browsing session and returns the 1st matching file name.

NAME = **FILE_FINDFIRST** (PATH)

input

PATH :	ANY_STRING	path and name for the search; the path specifies the location where files have to be searched; the name specifies the match key for the files to be returned; can include all the usual wildcards ('*', '?'); wildcards are allowed in the name part only, not in the path; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output

NAME :	WSTRING	the name of the 1st file found; an EMPTY string is returned if there are no matching files; both files and folders can be returned; conventional '.' and '..' folders are ignored; 'special' files (such as fifo, blocks, devices, sockets, and all the different elements that Linux adds to the file system) are ignored

Only one browsing session can exist at any given time.
This function can be used even if an old browsing session is already in progress: in this case the old one is automatically closed first (see **FILE_FINDCLOSE** for explicit close requests).

When this function returns a valid file name, the following variables can be used to retrieve additional information about the browsed entry:

FILE_FOUNDNAME	replicates the name of the last browsed file
FILE_FOUNDISDIR	states whether the last retrieved name is a directory (TRUE) or a file (FALSE)
FILE_FOUNDSIZE	the size of the last browsed file
FILE_FOUNDTIME	the write time of the last browsed file

Note that the given variables are meant to retain the values coming from the last successful browsing function invocation; in case of errors or empty strings returned by **FILE_FINDFIRST** or **FILE_FINDNEXT**, these variables values will remain unchanged.

example

```
VAR
   filename : WSTRING [256];
END_VAR;

filename := FILE_FINDFIRST (searchpath + '*');
WHILE (filename <> '') DO
   IF (FILE_FOUNDISDIR) THEN
      // handle the matching directory
   ELSE
      // handle the matching file
      // do whatever needed with FILE_FOUNDSIZE, FILE_FOUNDTIME, FILE_FOUNDNAME
   END_IF;
   filename := FILE_FINDNEXT ();
END_WHILE;
FILE_FINDCLOSE ();
```

FILE_FINDNEXT

Proceeds with the files browsing session in progress and return the next matching file name.

NAME = **FILE_FINDNEXT** ()

output

NAME :	WSTRING	the name of the next file found; an EMPTY string is returned if there are no more matching files; both files and folders can be returned; conventional '.' and '..' folders are ignored; 'special' files (such as fifo, blocks, devices, sockets, and all the different elements that Linux adds to the file system) are ignored

Can only be used if there actually is a session in progress: a browsing session must have already been opened with (a successful) **FILE_FINDFIRST** (and not closed yet with **FILE_FINDCLOSE**).

When this function returns a valid file name, the following variables can be used to retrieve additional information about the browsed entry:

FILE_FOUNDNAME	replicates the name of the last browsed file
FILE_FOUNDISDIR	states whether the last retrieved name is a directory (TRUE) or a file (FALSE)
FILE_FOUNDSIZE	the size of the last browsed file
FILE_FOUNDTIME	the write time of the last browsed file

Note that the given variables are meant to retain the values coming from the last successful browsing function invocation; in case of errors or empty strings returned by **FILE_FINDFIRST** or **FILE_FINDNEXT**, these variables values will remain unchanged.

See a usage *example* given along with the **FILE_FINDFIRST** description.

FILE_FINDCLOSE

Closes the files browsing session currently in progress.

FILE_FINDCLOSE ()

Can only be used if there actually is a session in progress: a browsing session must have already been opened with (a successful) **FILE_FINDFIRST**.

See a usage *example* given along with the **FILE_FINDFIRST** description.

FILE_AVAILABLESPACE

Retrieves the amount of available space on a given storage unit.

SPACE = **FILE_AVAILABLESPACE** (PATH)

input

PATH :	ANY_STRING	a path of any folder of the target storage unit; used to identify the unit itself; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output

SPACE :	ULINT	the available space (in bytes) on the identified storage unit

FILE_ABSOLUTEPATH

Returns an absolute file path and name, starting from a potentially relative path, or from a path containing system keys.

ABSOLUTE = **FILE_ABSOLUTEPATH** (PATH)

input

PATH :	ANY_STRING	path and name of a file or folder; the path can be either relative or (already) absolute, and can start with system keys such as $RESOURCES or $LOG; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules

output

ABSOLUTE :	WSTRING	the complete absolute path and name derived from the given one

The function is meant to resolve full path+name strings.
If only paths have to be treated, make sure to terminate them with '\' or '/' characters.
For example, if interested in knowing the plain content of a system key, the correct parameter would be something like "$LOG\" (not just "$LOG", that would not be recognized as a path segment).

example

```
_TRACE (FILE_ABSOLUTEPATH('C:\A\B\File.txt'));       // returns "C:\A\B\File.txt"
_TRACE (FILE_ABSOLUTEPATH('File.txt'));              // returns "C:\ESA\RTW\LOG\File.txt"
_TRACE (FILE_ABSOLUTEPATH('\File.txt'));             // returns "\File.txt"
_TRACE (FILE_ABSOLUTEPATH('$$RESOURCES\File.txt')); // returns "C:\ESA\RTW\RESOURCES\File.txt"
_TRACE (FILE_ABSOLUTEPATH('$$RESOURCES\'));          // returns "C:\ESA\RTW\RESOURCES\"
_TRACE (FILE_ABSOLUTEPATH('$$RESOURCES'));           // returns "C:\ESA\RTW\LOG\$RESOURCES" (see above)
_TRACE (FILE_ABSOLUTEPATH('\'));                     // returns "\"
_TRACE (FILE_ABSOLUTEPATH(''));                      // empty content not allowed
```

FILE_OPEN

Opens a file stream.
Returns a numeric identifier, expected in all subsequent calls to API functions for the same file stream.

FILE = **FILE_OPEN** (NAME, ACCESS, CREATION)

input		
NAME :	ANY_STRING	path and name of the file; must not be empty; see 'Files paths conventions' at the beginning of this section for notes about paths resolution rules
ACCESS :	ANY_UNSIGNED	a code stating the access mode; supported codes are: **0** (FILEREAD) enables read only **1** (FILEWRITE) enables write only **2** (FILEREADWRITE) enables both read and write
CREATION :	ANY_UNSIGNED	a code stating how to behave with regard to file creation; supported codes are: **0** (FILEDONTCREATE) never create a new file **1** (FILERESETANDCREATE) always restart with a new empty file **2** (FILECREATEIFNEEDED) keep existing files and create if needed
output		
FILE :	UDINT	a numeric identifier; used to identify the opened file in all the subsequent functions calls

The returned identifier is needed in the calls to any possible directive for the same file stream.
The identifier is unique among all the currently opened files. After closure (FILE_CLOSE) the identifier is free and might be reused for future streams.

This function affects the variable FILE_NUMBER, meant to count the number of open files.

About the meaning of the supported flags:

if ACCESS = FILEREAD
then the file can never be changed in any way, and never created or reset,
so the only possible value for CREATION is FILEDONTCREATE;

if ACCESS enables write as well (FILEWRITE or FILEREADWRITE),
then all the supported creation modes can be given;
in case of FILEDONTCREATE, the file must exist, and the operation fails otherwise; when opened the content is preserved;
in case of FILERESETANDCREATE, an existing file would be wiped out; regardless, a new empty file is recreated;
in case of FILECREATEIFNEEDED, if the file already exists its content is preserved; otherwise a new empty one is created.

FILE_CLOSE

Closes a file stream.
After a file has been closed, further calls to file stream API functions with the given identifier would result in failure.

FILE_CLOSE (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

The file must have been previously opened with a call to **FILE_OPEN**.
If needed, a cache flush is automatically invoked before closing; see **FILE_FLUSH** for an overview on the matter.

This function affects the variable **FILE_NUMBER**, meant to count the number of open files.

FILE_FLUSH

Flushes on disk all the pending cache buffers associated with the file content.

FILE_FLUSH (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

The file must have been previously opened with a call to **FILE_OPEN**.

The file cache buffers are implemented in order to keep write accesses to the flash memory at the minimum, to preserve the hardware durability.

Other than explicitly invoked with this method, files cache buffers can be automatically flushed in several situations as well:
- every call to any of the following functions automatically causes a flush:
  FILE_CLOSE, **FILE_REWIND**, **FILE_SEEK**, **FILE_READxxx** (any kind of read);
- if the script system variable **FILE_AUTOFLUSH** is set (TRUE, see the list of available variables below), then flushes are continuously done by the system: pending cache buffers are never maintained, so that the file is always synchronized with its content in the storage.

FILE_REWIND

Relocates the file pointer at the beginning of the file (the file pointer is the object that defines the starting position of the next read or write operation on the file stream).

FILE_REWIND (FILE)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

The file must have been previously opened with a call to **FILE_OPEN**.
If needed, a cache flush is automatically invoked before the movement; see **FILE_FLUSH** for an overview on the matter.

Meant for ease of use, it's the exact same thing as:

```
FILE_SEEK (file, 0, FILESTART);
```

(see **FILE_SEEK** below)

FILE_SEEK

Moves the file pointer to the given position (the file pointer is the object that defines the starting position of the next read or write operation on the file stream).

FILE_SEEK (FILE, OFFSET, START)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
OFFSET :	ANY_INT	offset of the movement, in bytes, from the given starting position
START :	ANY_UNSIGNED	identification of the movement starting position; supported codes are: **0** (**FILESTART**) start counting the offset from the beginning of the file **1** (**FILECURRENT**) start counting the offset from the current position **2** (**FILEEND**) start counting the offset from the end of the file

The file must have been previously opened with a call to **FILE_OPEN**.
If needed, a cache flush is automatically invoked before the movement; see **FILE_FLUSH** for an overview on the matter.

About the behaviour of the supported movements:
the OFFSET can be positive or negative; if positive, the pointer moves forward from the given starting position; if negative the pointer moves backward of the given amount of bytes;
if the START is **FILESTART**, then only positive offsets should be given; otherwise both positive and negative offsets are equally allowed;
offsets that take the file pointer to an absolute negative position (meaning movements of the pointer to a position that precedes the beginning of the file), are clipped to the beginning of the file itself;
offsets that take the file pointer to a position that exceeds the current file size (meaning movements of the pointer beyond the end of the file), are actually effective: subsequent read operations would fail with an EOF indication of course, but write requests would be allowed; in this last case the system would automatically fill the file room between its old end and the new write start (a warning though: different file systems could behave inconsistently with this regard, and the actual values of the filler bytes could be unpredictable; this practice should be avoided when this is not acceptable).

FILE_ISEOF

Checks whether the given file stream has reached its termination;
the condition is set after a read operation actively tried to read past the current end of the file.

EOF = **FILE_ISEOF** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
EOF :	BOOL	TRUE if the end has been reached; FALSE otherwise

The file must have been previously opened with a call to **FILE_OPEN**.

example

```
VAR
   fd   : UDINT;
   bval : BYTE;
END_VAR;

fd := FILE_OPEN ('FileName', FILEREAD, FILEDONTCREATE);
bval := FILE_READBYTE (fd);
WHILE (FILE_ISEOF(fd) = FALSE) DO
   // use <bval>
   bval := FILE_READBYTE (fd);
END_WHILE;
FILE_CLOSE (fd);
```

FILE_GETLENGTH

Retrieves the length of the current content of an opened file.

LENGTH = **FILE_GETLENGTH** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
LENGTH :	ULINT	current length (in bytes) of the file

The file must have been previously opened with a call to **FILE_OPEN**.
Not to be confused with the **FILE_GETSIZE**, used on regular unopened files.

FILE_GETPOSITION

Retrieves the position of the file pointer of an opened file stream;
the position is the offset in bytes, from the beginning of the file, where the next read or write operation would take place.

POSITION = **FILE_GETPOSITION** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
POSITION :	ULINT	the current position (in bytes from the beginning of the file) of the file pointer

The file must have been previously opened with a call to **FILE_OPEN**.

FILE_WRITEENCODING

Writes an encoding marker (UNICODE or uTF8) at the beginning of a (text) file.

FILE_WRITEENCODING (FILE, CODE)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
CODE :	ANY_UNSIGNED	the code of the new text encoding possible codes are: **0** (**FILEANSI**) **1** (**FILEUNICODE**) **2** (**FILEUTF8**)

The file must have been previously opened with a call to **FILE_OPEN**.

The UNICODE marker amounts to a conventional couple of bytes (0xFE, 0xFF) written at the beginning of a text file to declare that the content of the file will be encoded in UNICODE.
The UTF8 marker amounts to a conventional sequence of 3 bytes (0xBF, 0xBB, 0xEF) written at the beginning of a text file to declare that the content of the file will be encoded in UTF8.
Regarding the ANSI instead: there are no ANSI markers. Text files without a UNICODE or a UTF8 marker will be implicitly considered to be encoded in ANSI; calling this method with a **FILEANSI** CODE will simply do nothing.

This method is expected to be used on text files only, as soon as they are created, in order to be effective.
The markers must not be preceded by anything in the file; using this method when the file pointer is not positioned at the beginning of the file will result in failure.

After this method is used the file will be treated accordingly: all the string-oriented read and write operations will automatically handle the given encoding.
Interrogating the file with a **FILE_GETENCODING** (or later with a **FILE_READENCODING**) will allow to obtain the encoding code specified here.

Overwriting the marker of an already written file might result in unexpected behaviours of the involved data.

example

```
VAR
   fd : UDINT;
END_VAR;

fd := FILE_OPEN ('FileName', FILEWRITE, FILERESETANDCREATE);
FILE_WRITEENCODING (fd, FILEUNICODE);
...
FILE_CLOSE (fd);
```

FILE_READENCODING

Reads from the beginning of an opened file the marker of the used text encoding.

CODE = **FILE_READENCODING** (FILE)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output

CODE :	UDINT	the code of the recognized text encoding possible returned codes are: **0** (**FILEANSI**) **1** (**FILEUNICODE**) **2** (**FILEUTF8**)

The file must have been previously opened with a call to **FILE_OPEN**.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

This method is expected to be used on text files only, as soon as they are opened for read, in order to be effective.
The marker is expected to exist at the beginning of text files only; using this method when the file pointer is not positioned at the beginning of the file will result in failure.
After the reading, the marker has been taken into account and skipped by the file pointer; this means the file pointer will be set on the 1st byte of ANSI files, on the 3rd byte of UNICODE files, and on the 4th byte of UTF8 files.

After this method is used and the file is recognized to be designed in the declared way, the file content will be treated accordingly: all the string-oriented read and write operations will automatically handle the given encoding.

example

```
VAR
   fd : UDINT;
END_VAR;

fd := FILE_OPEN ('FileName', FILEREAD, FILEDONTCREATE);
FILE_READENCODING (fd);
...
FILE_CLOSE (fd);
```

FILE_SETENCODING

Forces a new encoding type for an opened text file.
The new code doesn't affect the content of the file in any way (nothing is read or written): no initial encoding marker is involved in the operation.

FILE_SETENCODING (FILE, CODE)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
CODE :	ANY_UNSIGNED	the code of the new text encoding possible codes are: **0** (**FILEANSI**) **1** (**FILEUNICODE**) **2** (**FILEUTF8**)

The file must have been previously opened with a call to **FILE_OPEN**.

This instruction can be repeatedly executed for the same file; it allows the creation of files, with or without encoding marker, containing texts automatically encoded with heterogeneous styles.

FILE_GETENCODING

Obtains the identifier of the text encoding enabled on an opened file.
The encoding must have already been determined someway (with a FILE_WRITEENCODING in case of newly created files, or with a FILE_READENCODING on reopened files, or with a FILE_SETENCODING on any file). In absence of any of these calls, the file is considered to have a plain ANSI encoding.

CODE = **FILE_GETENCODING** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to FILE_OPEN

output		
CODE :	UDINT	the code of the recognized text encoding possible returned codes are: **0** (FILEANSI) **1** (FILEUNICODE) **2** (FILEUTF8)

The file must have been previously opened with a call to FILE_OPEN.

FILE_READBYTE

Reads a single byte from an opened file stream.

VALUE = **FILE_READBYTE** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
VALUE :	BYTE	the value of the acquired byte

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled).
An EOF condition is raised though: programmers should ALWAYS check the result of a **FILE_ISEOF** after EVERY read operation to know if the request actually succeeded and the obtained value is from the file. In failure cases the returned VALUE should be 0, but programmers must NOT rely on this information to detect EOF conditions.

FILE_READWORD

Reads a single word (2 bytes) from an opened file stream.

VALUE = **FILE_READWORD** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
VALUE :	WORD	the value of the acquired word

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled).
An EOF condition is raised though: programmers should ALWAYS check the result of a **FILE_ISEOF** after EVERY read operation to know if the request actually succeeded and the obtained value is from the file. In failure cases the returned VALUE should be 0, but programmers must NOT rely on this information to detect EOF conditions.
If the EOF happened halfway, the return might come up with a 'partial' value set.

FILE_READDWORD

Reads a double word (4 bytes) from an opened file stream.

VALUE = **FILE_READDWORD** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
VALUE :	DWORD	the value of the acquired double word

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled).
An EOF condition is raised though: programmers should ALWAYS check the result of a **FILE_ISEOF** after EVERY read operation to know if the request actually succeeded and the obtained value is from the file. In failure cases the returned VALUE should be 0, but programmers must NOT rely on this information to detect EOF conditions.
If the EOF happened halfway, the return might come up with a 'partial' value set.

FILE_READLWORD

Reads a long word (8 bytes) from an opened file stream.

VALUE = **FILE_READLWORD** (FILE)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output		
VALUE :	LWORD	the value of the acquired long word

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled).
An EOF condition is raised though: programmers should ALWAYS check the result of a **FILE_ISEOF** after EVERY read operation to know if the request actually succeeded and the obtained value is from the file. In failure cases the returned VALUE should be 0, but programmers must NOT rely on this information to detect EOF conditions.
If the EOF happened halfway, the return might come up with a 'partial' value set.

FILE_READBUFFER

Reads a buffer in form of an array of bytes from an opened file stream.

VALUE = **FILE_READBUFFER** (FILE, LENGTH)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
LENGTH :	ANY_UNSIGNED	the number of bytes to read

output

VALUE :	ANY	values of the acquired buffer; the buffer is returned as an array of BYTEs, with the number of elements equal to the requested LENGTH

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled).
An EOF condition is raised though: programmers should ALWAYS check the result of a **FILE_ISEOF** after EVERY read operation to know if the request actually succeeded and the obtained value is from the file. In failure cases the returned VALUE should be an array filled with 0s, but programmers must NOT rely on this information to detect EOF conditions.

In these cases, if the EOF happened halfway, the returned buffer might be partially filled: the first bytes might contain the last piece of file, while the last bytes might remain untouched (filled with 0s).
The function **FILE_GETREADLEN** can be used to retrieve the number of bytes acquired with the last read directive: it will return the number of bytes actually stored in the **FILE_READBUFFER** output array.

See a usage *example* given along with the **FILE_GETREADLENGTH** description.

FILE_READSTRING

Reads a string of a given maximum length from an opened file stream.

VALUE = **FILE_READSTRING** (FILE, LENGTH)

input		
FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
LENGTH :	ANY_UNSIGNED	maximum length (in characters) of the read string

output		
VALUE :	ANY_STRING	the acquired string; could be either a simple STRING (in case of ANSI encoding) or a WSTRING (in case of UNICODE or UTF8 encoding)

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

The read string is expected to be stored in the file according to the known encoding mode; the proper number of bytes per character is acquired. The given string LENGTH limit is expressed in characters, not bytes.
The acquisition from the file stream terminates after the maximum number of characters have been read, or when the end of file has been reached.
The given string length is a maximum; it is possible to obtain valid shorter strings if the end of file is encountered.
This function should be treated carefully: since the operation doesn't take into account the actual values of the characters acquired, it is possible to read even characters that would normally break the string or make it inconsistent (for example the returned string could contain new lines, or could contain NUL characters that could make it unusable). It is programmer responsibility to use the function in environment where these conditions are controlled or acceptable.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled). The returned string together with the EOF condition (see **FILE_ISEOF**) can be used by the programmer to properly detect the EOF and handle file acquisition loops.
If the file pointer starts still within the boundaries of the file content, and the given string length limit takes it past the end, then a shorter string is returned and the EOF condition flag is raised;
if the file pointer is exactly at the end of the file, then the function returns an empty string, and the EOF condition flag is raised.

FILE_READLINE

Reads a string of a given maximum length from an opened file stream.
Almost identical to the **FILE_READSTRING** above, if not for the fact that the strings acquisitions automatically terminate when suitable line terminators are read from the file.

VALUE = **FILE_READLINE** (FILE, LENGTH)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**
LENGTH :	ANY_UNSIGNED	maximum length (in characters) of the read string

output

VALUE :	ANY_STRING	the acquired string; could be either a simple STRING (in case of ANSI encoding) or a WSTRING (in case of UNICODE or UTF8 encoding)

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for reading.
If needed, a cache flush is automatically invoked before the read; see **FILE_FLUSH** for an overview on the matter.

The read string is expected to be stored in the file according to the known encoding mode; the proper number of bytes per character is acquired. The given string length limit is expressed in characters, not bytes.
The acquisition from the file stream terminates after the maximum number of characters have been read, or when a line terminator has been read, or when the end of file has been reached (whichever condition comes first).
The given string length is a maximum; it is possible to obtain valid shorter strings if the end of file or a line termination are encountered.

About the line termination characters, all the following characters, or combination of characters, are considered valid terminations:
- 0x00 (NUL)
- 0x0A (a lone LF)
- 0x0D (a lone CR)
- 0x0A 0x0D (LF+CR)
- 0x0D 0x0A (CR+LF)

The termination (new line) characters are NOT included in the returned strings.

In case of reading past the end of the file, the function doesn't formally fail (no standard error is raised so that the script can continue regardless the errors management model enabled). The returned string together with the EOF condition (see **FILE_ISEOF**) can be used by the programmer to properly detect the EOF and handle file acquisition loops.
If the file pointer starts still within the boundaries of the file content, and the given string length limit takes it past the end, then a shorter string is returned and the EOF condition flag is raised;
if the file pointer is exactly at the end of the file, then the function returns an empty string, and the EOF condition flag is raised.

FILE_WRITE

Writes a value (any kind of value) in an opened file stream.

FILE_WRITE (FILE, VALUE [, SIZE])

input

FILE :	UDINT		file identifier returned by a previous call to **FILE_OPEN**
VALUE :	ANY		value to be written in the file
SIZE :	ANY_UNSIGNED	[OPTIONAL]	size (in bytes) of the part VALUE written in the file; can be used to limit the number of bytes actually written in cases where the data is passed in buffers (likely arrays) that might either be full or not; if this parameter is missing, then the whole VALUE is written; giving a SIZE bigger than the passed VALUE has no effect: the write is limited to the actual VALUE size anyway; this parameter is ignored in case of strings or arrays of strings, where each single element value can only be limited by its internal terminator

The file must have been previously opened with a call to **FILE_OPEN**.
The file must be opened for writing.

The function writes in the file exactly the number of bytes required by the type of the given value.
Any type of value can be passed to the function, included arrays, structures and strings.
Note that the structures are binary types, so even if they contain some form of string field, these are not converted with text encoding.
Plain strings and arrays of strings are managed instead: their value is converted according to the enabled text encoding.

The file stream might be automatically flushed if the **FILE_AUTOFLUSH** property demands so. Otherwise written values might remain in memory cache buffers for an undefined amount of time, until enough data has been bufferized, or until a flush is required, either by explicit calls or by implicit automatic mechanics. See **FILE_FLUSH** for details on the matter.

FILE_GETREADLENGTH

Retrieves the number of bytes that were acquired with the last call to a read function.

LENGTH = **FILE_GETREADLENGTH** (FILE)

input

FILE :	UDINT	file identifier returned by a previous call to **FILE_OPEN**

output

LENGTH :	ANY_UNSIGNED	file identifier returned by a previous call to **FILE_OPEN**

The file must have been previously opened with a call to **FILE_OPEN**.

The returned value is affected by calls to any of the available READ methods (**FILE_READBYTE**, **FILE_READWORD**, **FILE_READDWORD**, **FILE_READLWORD**, **FILE_READBUFFER**, **FILE_READSTRING**, **FILE_READLINE**).

This function is designed to be used in conjunction with **FILE_ISEOF** checks in loop situations: data can be repeatedly extracted in blocks from files, while continuously checking for the EOF and accessing only the portion of data actually retrieved. In particular, usage with **FILE_READBUFFER** is effective, since the mechanic above allows for acquisitions of data in large blocks with the ability to recognize the end and the size of the last piece of file.

example

```
VAR
   fd    : UDINT;
   index : LINT;
   block : ARRAY [100] OF BYTE;
END_VAR;

fd := FILE_OPEN ('FileName', 0, 0);

REPEAT
   block := FILE_READBUFFER (fd, 100);  // size must match the array declaration
   FOR index := 0 TO (FILE_GETREADLENGTH(fd) - 1) DO  // better used with STATIC_FOR to avoid stress
      // do something with <block[index]>
   END_FOR;
UNTIL (FILE_ISEOF(fd));
END_REPEAT;

FILE_CLOSE (fd);
```

< VARIABLES >

The following are the variables usable to share information and directives for the files management:

FILE_NUMBER
type UDINT
access R
gives the number of files currently opened by scripts

FILE_AUTOFLUSH
type BOOL
access R/W
states whether writes in file streams have to be immediately and automatically flushed on the storage (TRUE), or cache buffers can be used to minimize the storage write accesses (FALSE);
see FILE_FLUSH for an overview about file flush needs

FILE_FOUNDNAME
type WSTRING
access R
replicates the name of the last file (or directory) successfully browsed and returned by a FILE_FINDFIRST or FILE_FINDNEXT;
the returned information is updated (only) every time a browsing function succeeds; failed calls or calls returning empty names to notify the end of the elements list don't affect the storage, meant to preserve the last valid entry;
the information persists even after the browsing session is closed

FILE_FOUNDSIZE
type ULINT
access R
gives the size (in bytes) of the last file (or directory) successfully browsed and returned by a FILE_FINDFIRST or FILE_FINDNEXT;
directories have a size of 0 bytes;
see FILE_FOUNDNAME for hints

FILE_FOUNDTIME
type LDT
access R
gives the last write time of the last file (or directory) successfully browsed and returned by a FILE_FINDFIRST or FILE_FINDNEXT;
times are returned in the form (local or UTC) given by the file system in use; normally UTC is the way;
see FILE_FOUNDNAME for hints

FILE_FOUNDISDIR
type BOOL
access R
checks whether the last file (or directory) successfully browsed and returned by a FILE_FINDFIRST or FILE_FINDNEXT, is a file or a directory:
FALSE means it's a file,
TRUE means it's a directory;
see FILE_FOUNDNAME for hints

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
FILEREAD	0	**FILE_OPEN**
FILEWRITE	1	**FILE_OPEN**
FILEREADWRITE	2	**FILE_OPEN**
FILEDONTCREATE	0	**FILE_OPEN**
FILERESETANDCREATE	1	**FILE_OPEN**
FILECREATEIFNEEDED	2	**FILE_OPEN**
FILESTART	0	**FILE_SEEK**
FILECURRENT	1	**FILE_SEEK**
FILEEND	2	**FILE_SEEK**
FILEANSI	0	**FILE_WRITEENCODING**, **FILE_READENCODING**, **FILE_SETENCODING**, **FILE_GETENCODING**
FILEUNICODE	1	**FILE_WRITEENCODING**, **FILE_READENCODING**, **FILE_SETENCODING**, **FILE_GETENCODING**
FILEUTF8	2	**FILE_WRITEENCODING**, **FILE_READENCODING**, **FILE_SETENCODING**, **FILE_GETENCODING**
FILECSV	0	**$$$_EXPORT**, **$$$_RESET**
FILEPDF	1	**$$$_EXPORT**, **$$$_RESET**

6. COMMON - SERIAL

COM_OPEN

Opens a communication channel on a serial port.

PORT = **COM_OPEN** (COM, BAUD, DATA, PARITY, STOP)

input		
COM :	ANY_INT	the number of the COM port that must be opened; must be a COM# available on the server machine
BAUD :	ANY_INT	the baud rate for the communication channel; must be the exact number of bps; the possible values are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
DATA :	ANY_INT	number of data bits; the following values and constants can be used: **5** (COMDATA5) **6** (COMDATA6) **7** (COMDATA7) **8** (COMDATA8)
PARITY :	ANY_INT	type of parity check; the following values and constants can be used: **0** (COMPARITYNONE) **1** (COMPARITYODD) **2** (COMPARITYEVEN) **3** (COMPARITYMARK) **4** (COMPARITYSPACE)
STOP :	ANY_INT	number of stop bits; the following values and constants can be used: **0** (COMSTOP1) **1** (COMSTOP1_5) (not supported in **Linux**) **2** (COMSTOP2)

output		
PORT :	UDINT	a server-generated identifier for the opened port; this ID is expected to be used in any further call to COM functions aimed to use the same port

The returned PORT identifier is needed in the calls to any further directive for the same COM port.
The identifier is unique among all the currently opened ports. After closure (COM_CLOSE) the identifier is free and might be reused.

Serial ports are opened by default to behave in a standard way; in particular, hardware flow controls are disabled. To change the state of automatic flow controls, use the function COM_FLOW.

This function affects the variable COM_NUMBER, meant to count the number of open serial connections.

COM_CLOSE

Closes an opened serial port.

COM_CLOSE (PORT)

input

PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

The serial port must have been previously opened with a call to **COM_OPEN**.

After a port has been closed, further calls to COM API functions with the given identifier would result in failure.

After a port has been closed, its identifier is released and might be reused as identifier for ports opened in the future.

This function affects the variable **COM_NUMBER**, meant to count the number of open serial connections.

COM_FLOW

Sets up the flow controls of an opened serial port.

COM_FLOW (PORT, RTS, CTS, DTR, DSR)

input		
PORT :	UDINT	the port identifier; obtained by a previous call to **COM_OPEN**
RTS :	ANY_INT	RTS flow control; the following values and constants can be used: **0** (**COMRTSDISABLE**) **1** (**COMRTSENABLE**) **2** (**COMRTSHANDSHAKE**) **3** (**COMRTSTOGGLE**)
CTS :	ANY_INT	CTS handshaking; the following values and constants can be used: **0** (**COMCTSOFF**) **1** (**COMCTSON**)
DTR :	ANY_INT	DTR flow control; the following values and constants can be used: **0** (**COMDTRDISABLE**) **1** (**COMDTRENABLE**) **2** (**COMDTRHANDSHAKE**)
DSR :	ANY_INT	DSR handshaking; the following values and constants can be used: **0** (**COMDSROFF**) **1** (**COMDSRON**)

As soon as the serial port is opened (**COM_OPEN**) all the automatic flow controls are disabled by default.
Use this function to enable the RTS/CTS or DTR/DSR flow control as needed.

Note: in **Linux** platforms only RTS/CTS flow controls are supported.
Also, only a plain enable/disable is allowed (there is no behaviour management at all).
Therefore the only meaningful parameter here is RTS, and only the values **COMRTSDISABLE** and **COMRTSENABLE** are actually used. Everything else is simply ignored.

COM_ISOPEN

Checks whether a port with given identifier is currently open.

STATE = **COM_ISOPEN** (PORT)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output		
STATE :	BOOL	TRUE if the port is currently open; FALSE otherwise

The serial port identifier is supposed to be one obtained by a previous call to a **COM_OPEN**.
The function will tell whether the port is still open or not.
Invalid IDs will be undistinguishable from closed ports.

COM_DATALENGTH

Retrieves the number of bytes currently available for reading from an opened serial port.

LENGTH = **COM_DATALENGTH** (PORT)

input

PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output

LENGTH :	UDINT	the number of available bytes

The serial port must have been previously opened with a call to **COM_OPEN**.

The function won't block and won't wait for anything: it will just return the number of bytes already pending in the port buffer.

COM_READBYTE

Reads a single byte from an opened serial port.

BYTE = **COM_READBYTE** (PORT, TIMEOUT)

input

PORT :	UDINT	the port identifier, obtained by a previous call to COM_OPEN
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet

output

BYTE :	BYTE	the value of the acquired byte

The serial port must have been previously opened with a call to COM_OPEN.

If the function successfully reads a byte from the serial port, its value is returned, and the variable COM_RXLENGTH will state that **1** byte has been read.
If the function succeeds in the serial port management, but nothing is found to be available for reading within the given timeout, then the function returns 0, and the variable COM_RXLENGTH will state that **0** bytes have been read.

In other words, script error states can be used to check whether the port management failed at system level, but the variable COM_RXLENGTH should be used to see if anything was actually read from the port.
Another way to handle the same issue: this function could be used together with the COM_DATALENGTH: availability of bytes in buffer could be checked beforehand, so that the read would be called only if actually necessary.

example

```
VAR
   com : UDINT;
   cbt : BYTE := 0;
   eos : BYTE := 120;   // the example loop will stop when this byte is received
END_VAR;

com := COM_OPEN (1, 115200, COMDATA8, COMPARITYNONE, COMSTOP1);
WHILE cbt <> eos DO
   // 1st way: test availability using COM_DATALENGTH function
      SLEEP (10);
      IF COM_DATALENGTH(com) > 0 THEN
         cbt := COM_READBYTE (com, 0);
         // do whatever needed with the acquired <cbt>
      END_IF;
   // 2nd way: test availability using COM_RXLENGTH variable
      cbt := COM_READBYTE (com, 10);
      IF COM_RXLENGTH > 0 THEN
         // do whatever needed with the acquired <cbt>
      END_IF;
END_WHILE;
COM_CLOSE (com);
```

COM_READBUFFER

Reads a sequence of bytes from an opened serial port.

VALUE = **COM_READBUFFER** (PORT, SIZE, TIMEOUT)

input

PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**
SIZE :	ANY_INT	the maximum number of bytes to get from the port; this value is currently limited to **65536** bytes (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	ANY	despite the generic declaration, the result is always an array of bytes; the size of the array is defined by the input parameter SIZE

The serial port must have been previously opened with a call to **COM_OPEN**.

In case of successful execution, the function returns an array of bytes with the given SIZE. Not necessarily all the bytes of the array have to be filled up with read bytes: some might be reset to 0.
The variable **COM_RXLENGTH** will count the number of bytes read and returned.

The given SIZE is the number of elements of the returned array, and the maximum number of bytes acquired from the serial port. The key word is 'maximum': the function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned array is completely reset, and the variable **COM_RXLENGTH** is set to 0;
- if any number of bytes within the given SIZE are found to be available, then all these bytes are acquired and returned, the remaining part of the array remains reset, and the variable **COM_RXLENGTH** is set to that exact number of bytes;
- if more than SIZE bytes are available, then only the first SIZE ones are read and returned; the variable **COM_RXLENGTH** is set to SIZE.

At any given time this function should be able to read as many bytes as stated by **COM_DATALENGTH**.

example

```
VAR
   com, cbt : UDINT;    car : ARRAY [100] OF BYTE;
END_VAR;

com := COM_OPEN (1, 115200, COMDATA8, COMPARITYNONE, COMSTOP1);
WHILE <loop is needed> DO
   car := COM_READBUFFER (com, 100, 1000);  // wait up to 1 second to get something to read
   IF COM_RXLENGTH > 0 THEN
      FOR cbt := 0 TO COM_RXLENGTH-1 DO
         // do whatever needed with the acquired bytes <car[cbt]>
      END_FOR;
   END_IF;
END_WHILE;
COM_CLOSE (com);
```

COM_WRITE

Writes data on an opened serial port.

COM_WRITE (PORT, VALUE [, SIZE])

input

PORT :	UDINT		the port identifier, obtained by a previous call to **COM_OPEN**
VALUE :	ANY		the value that has to be sent on the serial port; this could be a value of any type; the transmitted data will be the exact binary image of the given value; the programmer is supposed to know the binary format of the used types
SIZE :	ANY_INT	[OPTIONAL]	size (in bytes) of the part of VALUE sent on the serial port; can be used to limit the number of bytes actually transmitted in cases where the data is passed in buffers (likely arrays) that might either be full or not; if this parameter is missing, then the whole VALUE is written; giving a SIZE bigger than the passed VALUE has no effect: the transmission is limited to the actual VALUE size anyway

The serial port must have been previously opened with a call to **COM_OPEN**.

The function sends exactly the number of bytes required by the type of the given value.
Any type of value can be passed to the function, included arrays, structures and strings.
The programmer must be aware of the exact format, size and endianity of the written data.
Note that even strings follow the same rule: plain STRINGs are sent with 1 byte per character, while WSTRINGs are sent using 2 bytes per character.
A degree of management on plain strings is implemented actually: whereas arrays of strings are sent like binary chunks of data (one byte for each byte of data buffer), plain strings are in fact terminated at their NUL terminator (see example below).

The variable **COM_TXLENGTH** will count the number of bytes sent by a successful execution of this function.

example

```
VAR
   com : UDINT;
   cbt : BYTE;
   clw : LWORD;
   cab : ARRAY [100] OF BYTE;
   cas : ARRAY [10] OF STRING [5];
   cst : WSTRING [10];
END_VAR;

com := COM_OPEN (1, 115200, COMDATA8, COMPARITYNONE, COMSTOP1);
COM_WRITE (com, cbt);                // 1 byte sent
COM_WRITE (com, clw);                // 8 bytes sent
COM_WRITE (com, cab);                // 100 bytes sent
COM_WRITE (com, cas);                // 10*(5+1) = 60 bytes sent
cst := 'abc'; COM_WRITE (com, cst);  // 3*2 = 6 bytes sent
COM_CLOSE (com);
```

COM_CLEAR

Clears buffers content and states of an opened serial port.

COM_CLEAR (PORT)

input

PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

The serial port must have been previously opened with a call to **COM_OPEN**.

Error states will be reset, output buffers will be flushed, and input buffers will be cleared.
Newly opened ports are forcefully set in a clear state.

COM_GETCTS

Retrieves the current state of the CTS signal of an opened serial port.

STATE = **COM_GETCTS** (PORT)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output		
STATE :	BOOL	the current signal state

The serial port must have been previously opened with a call to **COM_OPEN**.

COM_GETDSR

Retrieves the current state of the DSR signal of an opened serial port.

STATE = **COM_GETDSR** (PORT)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output		
STATE :	BOOL	the current signal state

The serial port must have been previously opened with a call to **COM_OPEN**.

COM_GETRING

Retrieves the current state of the RING signal of an opened serial port.

STATE = **COM_GETRING** (PORT)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output		
STATE :	BOOL	the current signal state

The serial port must have been previously opened with a call to **COM_OPEN**.

COM_GETRLSD

Retrieves the current state of the RLSD signal of an opened serial port.

STATE = **COM_GETRLSD** (PORT)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**

output		
STATE :	BOOL	the current signal state

The serial port must have been previously opened with a call to **COM_OPEN**.

Note: in **Linux** platforms this signal is not supported.

COM_SETRTS

Sets the state of the RTS signal of an opened serial port.

COM_SETRTS (PORT, STATE)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**
STATE :	ANY_INT	the new signal state; ideally only boolean values should be used; any non-zero value is interpreted as a 'true'

The serial port must have been previously opened with a call to **COM_OPEN**.

COM_SETDTR

Sets the state of the DTR signal of an opened serial port.

COM_SETDTR (PORT, STATE)

input		
PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**
STATE :	ANY_INT	the new signal state; ideally only boolean values should be used; any non-zero value is interpreted as a 'true'

The serial port must have been previously opened with a call to **COM_OPEN**.

COM_SET485

Enables the 485 mode on an opened serial port.

COM_SET485 (PORT, MODE)

input

PORT :	UDINT	the port identifier, obtained by a previous call to **COM_OPEN**
MODE :	ANY_INT	the port type mode; possible codes are: **0** (**COMMODE232**) : 485 is disabled **1** (**COMMODE485**) : 485 is enabled

The serial port must have been previously opened with a call to **COM_OPEN**.

< VARIABLES >

The following are the variables usable to share information and directives for the serial ports management:

COM_NUMBER
type UDINT
access R
gives the number of serial ports currently opened by scripts

COM_RXLENGTH
type UDINT
access R
gives the number of bytes acquired by the last successful execution of a read function (see COM_READBYTE and COM_READBUFFER);
failed calls won't affect this variable;
the given value could be 0 in case the invoked function found nothing to read from the serial port

COM_TXLENGTH
type UDINT
access R
gives the number of bytes transmitted by the last successful execution of a write function (see COM_WRITE);
failed calls won't affect this variable

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
COMDATA5	5	COM_OPEN
COMDATA6	6	COM_OPEN
COMDATA7	7	COM_OPEN
COMDATA8	8	COM_OPEN
COMPARITYNONE	0	COM_OPEN
COMPARITYODD	1	COM_OPEN
COMPARITYEVEN	2	COM_OPEN
COMPARITYMARK	3	COM_OPEN
COMPARITYSPACE	4	COM_OPEN
COMSTOP1	0	COM_OPEN
COMSTOP1_5	1	COM_OPEN
COMSTOP2	2	COM_OPEN
COMRTSDISABLE	0	COM_FLOW
COMRTSENABLE	1	COM_FLOW
COMRTSHANDSHAKE	2	COM_FLOW
COMRTSTOGGLE	3	COM_FLOW
COMCTSOFF	0	COM_FLOW
COMCTSON	1	COM_FLOW
COMDTRDISABLE	0	COM_FLOW
COMDTRENABLE	1	COM_FLOW
COMDTRHANDSHAKE	2	COM_FLOW
COMDSROFF	0	COM_FLOW
COMDSRON	1	COM_FLOW
COMMODE232	0	COM_SET485
COMMODE485	1	COM_SET485

7. COMMON - ETHERNET

ETH_IP

Creates an IP address.

IP = **ETH_IP** (NUMERIC)
IP = **ETH_IP** (STRING)
IP = **ETH_IP** (COMP1, COMP2, COMP3, COMP4)

input		
NUMERIC :	ANY_UNSIGNED	unique numeric field with IP address; only UDINT and DWORD types are actually allowed in this case; the 1st IP component is expected to be in the MSB of the given value
input		
STRING :	ANY_STRING	the IP address is given in string form; the string must be formatted as "IP1.IP2.IP3.IP4"
input		
COMP# :	ANY_INT	the 4 components of the IP address are given separately in numeric form; all the 4 components must be given, in the correct order (IP1, IP2, IP3, IP4)
output		
IP :	UDINT	the normalized IP address

This function is used to normalize the format of IP addresses used with all the functions available for ethernet management.
All the implemented functions expect IP addresses to be specified as UDINT values in the conventional <in_addr> numeric format, with the 1st component of the address in the LSB and the last component in the MSB of the address value.
Programmers though could find more 'friendly' ways to specify the addresses to better fit their needs. This family of **ETH_IP** functions allows the creation of normalized addresses starting from a number of different formats.

example

```
VAR
   ipadd : UDINT;
END_VAR;

ipadd := ETH_IP ("127.0.0.1");      // numeric result = 0x0100007F
ipadd := ETH_IP (127, 0, 0, 1);     // numeric result = 0x0100007F
ipadd := ETH_IP (16#7F000001);      // numeric result = 0x0100007F
```

ETH_GETIP

Converts a normalized IP address in different forms.

IPSTR = **ETH_GETIP** (IP)

input		
IP :	UDINT	a normalized IP address; see **ETH_IP** for information about IPs normalization

output		
IPSTR :	STRING	the given IP formatted as string; the returned string is of course in the form "IP1.IP2.IP3.IP4"

This function doesn't only convert the IP address in a string: it contemporarily split the address in its 4 numeric components and store them in the 4 variables **ETH_IP1**, **ETH_IP2**, **ETH_IP3**, **ETH_IP4**.
These variables are only updated by successful execution of this function; in case of errors they are left unchanged.

ETH_PING

Sends a PING (ICMP echo request) to a remote device of given IP address.

ECHO = **ETH_PING** (IP, TIMEOUT [, ATTEMPTS])

input

IP :	UDINT		normalized IP address of the remote device
TIMEOUT :	ANY_INT		maximum time to wait for the echo; given in milliseconds
ATTEMPTS :	ANY_INT	[OPTIONAL]	maximum number of PING attempts; the system automatically retries to send its request until a successful answer is received

output

ECHO :	BOOL	TRUE if the echo answer has been received within the given TIMEOUT; FALSE if nothing has been received for the whole TIMEOUT duration

In case of ping error (a successful execution with a FALSE result) the variable **ETH_ERROR** might give further details about what could have gone wrong.

example

```
VAR
   ethres  : BOOL;
   ethscan : ARRAY [256] OF BOOL;
   ethidx  : UINT;
   ethx, ethy : UINT;
   ethmap  : ARRAY [16] OF STRING [32];
END_VAR;

// Build up a map of reachable devices
ethscan[0] := FALSE;
FOR ethidx := 1 TO 255 DO
   ethscan[ethidx] := ETH_PING (ETH_IP (172, 19, 5, ethidx), 200, 1);
END_FOR;

// Show the map as a string matrix
FOR ethy := 0 TO 15 DO                          // 16 rows of 16 chars each
   ethidx := 0;                                 // counts the devices in each row
   ethmap[ethy] := '';
   FOR ethx := 0 TO 15 DO
      IF ethscan[ethy*16+ethx] THEN
         ethmap[ethy] := ethmap[ethy] + '*';  // * char for each existing device
         ethidx := ethidx + 1;
      ELSE
         ethmap[ethy] := ethmap[ethy] + '.';  // . char for each missing device
      END_IF;
   END_FOR;
   ethmap[ethy] := ethmap[ethy] + '   ' + ANY_TO_STRING (ethidx);
   _TRACE (ethmap[ethy]);
END_FOR;
```

ETH_TCPC_OPEN

Opens an ethernet communication channel connecting as a TCP client to a listening TCP server.

ETHID = **ETH_TCPC_OPEN** (IP, PORT [, LOCALPORT])

input			
IP :	UDINT		the (normalized) IP address of the remote server; it's the address (or one of many) the server is supposed to be listening from
PORT :	ANY_INT		the IP port opened by the remote server; it's the port the server is supposed to be listening from
LOCALPORT :	ANY_INT	[OPTIONAL]	used to force a fixed specific local IP port for the communication channel; if missing, the local port is automatically selected by the system among the available ones; for the same purpose, this local port could be explicitly specified as **ETHPORTAUTO**

output		
ETHID :	UDINT	the identifier of the created communication channel; it is defined as the index (base-0) of the ethernet manager instance associated to the opened channel; this identifier will have to be used by all the subsequent script functions (**ETH_TCPC_$$$**) related to this client

This function affects the variable **ETH_NUMBER**, meant to count the number of open ethernet channels.

The maximum number of ethernet channels managed by scripts is currently limited to **10**. Once the **ETH_NUMBER** reaches this limit, no more channels can be opened: new **ETH_$$$_OPEN** calls will fail.

ETH_TCPC_CLOSE

Closes an ethernet communication channel previously opened as a TCP client.

ETH_TCPC_CLOSE (ETHID)

input		
ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**

The targeted ETHID must be related to a TCP client.

This function affects the variable **ETH_NUMBER**, meant to count the number of open ethernet channels.

This function reports a successful result if the system is able to reset the channel and remove it from the managed ones; if, in the process, something goes wrong on the socket side, the variable **ETH_ERROR** might give details about the issue.

ETH_TCPC_GETIPLOCAL

Retrieves the IP address set up as local (client) address when the connection with a server was established.

IP = **ETH_TCPC_GETIPLOCAL** (ETHID)

input

ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**

output

IP :	UDINT	the (normalized) IP address of the local client; it's the local address used by the socket when the connection with the server was established

The targeted ETHID must be related to a TCP client.

As already mentioned, the returned IP address is in normalized (numeric) form.
Along with it, a couple of further variables are set up:
- **ETH_IPADDRESS** will replicate the returned IP address
- **ETH_IPPORT** will give the IP port associated with the communication channel

These variables are affected by every successful execution of this function; old values will be retained in case of errors. Note that several other ethernet functions are meant to affect them though (see their description in the <VARIABLES> section).

example

```
VAR
   IPadd : UDINT;
   IPport : UINT;
   ETHsvr, ETHcln : UDINT;
END_VAR

ETHsvr := ETH_TCPS_OPEN (ETHADDRESSANY, 2048);      // open a channel as server
ETHcln := ETH_TCPC_OPEN (ETH_IP(127,0,0,1), 2048);  // open a channel as client
// ETH_NUMBER : should state that there are 2 channels

IPadd := ETH_TCPC_GETIPLOCAL (ETHcln);
// IPadd and ETH_IPADDRESS should be 127.0.0.1  |  ETH_IPPORT is a port selected by the OS

IPadd := ETH_TCPC_GETIPSERVER (ETHcln);
// IPadd and ETH_IPADDRESS should be 127.0.0.1  |  ETH_IPPORT should be 2048

IPadd := ETH_TCPS_GETIPLOCAL (ETHsvr);
// IPadd and ETH_IPADDRESS should be 0.0.0.0    |  ETH_IPPORT should be 2048

IPadd := ETH_TCPS_GETIPCLIENT (ETHsvr, 0);
// IPadd and ETH_IPADDRESS should be 127.0.0.1  |  ETH_IPPORT is a port selected by the OS

ETH_TCPC_CLOSE (ETHcln);
ETH_TCPS_CLOSE (ETHsvr);
```

ETH_TCPC_GETIPSERVER

Retrieves the IP address of the remote server the TCP client connected to.

IP = **ETH_TCPC_GETIPSERVER** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
output		
IP :	UDINT	the (normalized) IP address of the remote server; it's the address that was explicitly given when the connection was requested by the function **ETH_TCPC_OPEN**

The targeted ETHID must be related to a TCP client.

As already mentioned, the returned IP address is in normalized (numeric) form.
Along with it, a couple of further variables are set up:
- **ETH_IPADDRESS** will replicate the returned IP address
- **ETH_IPPORT** will give the IP port associated with the communication channel

These variables are affected by every successful execution of this function; old values will be retained in case of errors. Note that several other ethernet functions are meant to affect them though (see their description in the <VARIABLES> section).

ETH_TCPC_DATALENGTH

Retrieves the number of bytes currently available for reading from a given channel.

LENGTH = **ETH_TCPC_DATALENGTH** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
output		
LENGTH :	UDINT	the number of available bytes, already readable from the given channel

The targeted ETHID must be related to a TCP client.

Note that even in case of successful execution, if information inconsistencies are found on the socket side, error codes might be notified by the **ETH_ERROR** variable.

ETH_TCPC_READBYTE

Reads a single byte of data from a given ethernet channel.

VALUE = **ETH_TCPC_READBYTE** (ETHID, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	BYTE	the value of the acquired byte

The targeted ETHID must be related to a TCP client.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired.
Being this function meant to read 1 only byte, the variable value will be 1 in case data was available for reading, or 0 otherwise.
In case of errors, the variable remains unaffected.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

example

```
VAR
   ethid : UDINT;
   length, index : UDINT;
   value : BYTE;
END_VAR

...
// Simple way with <DATALENGTH>
length := ETH_TCPC_DATALENGTH (ethid);
FOR index := 1 TO length DO
   value := ETH_TCPC_READBYTE (ethid, 0);
   // do whatever needed with the acquired <value>
END_FOR;
...
// Simple way with <ETH_RXLENGTH>
WHILE TRUE DO
   value := ETH_TCPC_READBYTE (ethid, 1);
   IF (ETH_RXLENGTH > 0) THEN
      // do whatever needed with the acquired <value>
   ELSE
      EXIT;
   END_IF;
END_WHILE;
```

ETH_TCPC_READBUFFER

Reads a whole buffer of data from a given ethernet channel.

VALUE = **ETH_TCPC_READBUFFER** (ETHID, SIZE, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
SIZE :	ANY_INT	the maximum number of bytes to get from the ethernet; this value is currently limited to **65536** bytes (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	ANY	the value of the acquired buffer; despite the generic declaration, this result is always an array of bytes; the size of the array is defined by the input parameter SIZE

The targeted ETHID must be related to a TCP client.

In case of successful execution, the function returns an array of bytes with the given SIZE. Not necessarily all the bytes of the array have to be filled up with read bytes: some might be reset to 0.
The variable **ETH_RXLENGTH** will count the number of bytes read and returned.

The given SIZE is the number of elements of the returned array, and the maximum number of bytes acquired from the ethernet. The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned array is completely reset, and the variable **ETH_RXLENGTH** is set to 0;
- if any number of bytes within the given SIZE are found to be available, then all these bytes are acquired and returned, the remaining part of the array remains reset, and the variable **ETH_RXLENGTH** is set to that exact number of bytes;
- if more than SIZE bytes are available, then only the first SIZE ones are read and returned; the variable **ETH_RXLENGTH** is set to SIZE.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPC_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_TCPC_READSTRING

Reads a whole buffer of data in string (short) form from a given ethernet channel.

VALUE = **ETH_TCPC_READSTRING** (ETHID, SIZE, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
SIZE :	ANY_INT	the maximum number of characters to get from the ethernet; this value is currently limited to **65536** characters (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	STRING	the value of the acquired string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a TCP client.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired (same as the number of characters).
The stored value could be anything between 0 and SIZE, depending on the amount of data available for reading.
In case of errors, the variable remains unaffected.

The given SIZE is the number of characters the returned string will be allocated for (and the maximum number of bytes acquired from the ethernet). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned. The acquisition terminates:
- when the timeout is reached,
- when the maximum number of characters has been read,
- when there is nothing more to read,
- when a NUL terminator is read.

In particular, about the given limits:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned string is empty, and the variable **ETH_RXLENGTH** is set to 0;
- if any number of characters within the given SIZE are found to be available, then all these characters are acquired and returned in the result string, the string is terminated accordingly, and the variable **ETH_RXLENGTH** is set to that exact number of bytes;
- if more than SIZE characters are available, then only the first SIZE ones are read and returned in the string; the variable **ETH_RXLENGTH** is set to SIZE.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPC_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong. Similarly, in case issues arise on the socket side after a part of string has been acquired already, then the function successfully returns what has been read in the output string, and let the variable **ETH_ERROR** give details about the error.

ETH_TCPC_READWSTRING

Reads a whole buffer of data in string (wide) form from a given ethernet channel.

VALUE = **ETH_TCPC_READWSTRING** (ETHID, SIZE, TIMEOUT)

input		
ETHID :	UDINT	the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
SIZE :	ANY_INT	the maximum number of characters to get from the ethernet; this value is currently limited to **32768** characters (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output		
VALUE :	WSTRING	the value of the acquired (wide) string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a TCP client.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired; note that this is NOT the same as the number of characters (with wide strings involved, each character is made of 2 bytes).
The stored value could be anything between 0 and SIZE*2, depending on the amount of data available for reading, and is supposed to always be an even value (multiple of 2, being the characters made of 2 bytes each).
In case of errors, the variable remains unaffected.

The given SIZE is the number of characters the returned string will be allocated for (the maximum number of bytes acquired from the ethernet will be twice as much). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned. The acquisition terminates:
- when the timeout is reached,
- when the maximum number of characters has been read,
- when there is nothing more to read,
- when a NUL terminator is read.

In particular, about the given limits:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned string is empty, and the variable **ETH_RXLENGTH** is set to 0;
- if any number of characters within the given SIZE are found to be available, then all these characters are acquired and returned in the result string, the string is terminated accordingly, and the variable **ETH_RXLENGTH** is set to that number of bytes (twice the number of characters);
- if more than SIZE characters are available, then only the first SIZE ones are read and returned in the string; the variable **ETH_RXLENGTH** is set to SIZE * 2.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPC_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong. Similarly, in case issues arise on the socket side after a part of string has been acquired already, then the function successfully returns what has been read in the output string, and let the variable **ETH_ERROR** give details about the error.

ETH_TCPC_WRITE

Writes data on an opened ethernet channel.

ETH_TCPC_WRITE (ETHID, VALUE [, SIZE])

input

ETHID :	UDINT		the identifier of an existing client channel; must be a valid identifier returned by a previous call to **ETH_TCPC_OPEN**
VALUE :	ANY		the value that has to be transmitted; this could be a value of any type; the transmitted data will be the exact binary image of the given value; the programmer is supposed to know the binary format of the used types
SIZE :	ANY_INT	[OPTIONAL]	size (in bytes) of the part of VALUE sent on the ethernet; can be used to limit the number of bytes actually transmitted in cases where the data is passed in buffers (likely arrays) that might either be full or not; if this parameter is missing, then the whole VALUE is written; giving a SIZE bigger than the passed VALUE has no effect: the transmission is limited to the actual VALUE size anyway

The targeted ETHID must be related to a TCP client.

The function transmits exactly the number of bytes required by the type of the given value.
Any type of value can be passed to the function, included arrays, structures and strings.
The programmer must be aware of the exact format, size and endianity of the written data.
Note that even strings follow the same rule: plain STRINGs are sent with 1 byte per character, while WSTRINGs are sent using 2 bytes per character.
A degree of management on plain strings is implemented actually: whereas arrays of strings are sent like binary chunks of data (one byte for each byte of data buffer), plain strings are in fact terminated at their NUL terminator.

The variable **ETH_TXLENGTH** will count the number of bytes transmitted by a successful execution of this function (failed executions will leave the variable unchanged).

In case of successful result, if the function couldn't write anything (**ETH_TXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

example

```
VAR
    cbt : BYTE;
    cab : ARRAY [100] OF BYTE;
    cas : ARRAY [10] OF STRING [5];
END_VAR;

...
ETH_TCPC_WRITE (ethid, cbt);      // ETH_TXLENGTH = 1 byte sent
ETH_TCPC_WRITE (ethid, cab);      // ETH_TXLENGTH = 100 bytes sent
ETH_TCPC_WRITE (ethid, cas);      // ETH_TXLENGTH = 10*(5+1) = 60 bytes sent
```

ETH_TCPS_OPEN

Creates a listening socket as TCP server.
Actual communication channels will be automatically created when clients submit their connection requests.

ETHID = **ETH_TCPS_OPEN** (IP, PORT)

input		
IP :	UDINT	the (normalized) IP address for the local listening socket; identifies the address of the ethernet port from which the server is expected to receive the clients connection requests; can be **ETHADDRESSANY** if the server is supposed to accept connections from any available ethernet port
PORT :	ANY_INT	the IP port opened by the server for its listening socket; this port is a mandatory specification (explicitly forbidden to be an **ETHPORTAUTO**): it's the port that the clients will have to use in their connection requests
output		
ETHID :	UDINT	the identifier of the created server channel; it is defined as the index (base-0) of the ethernet manager instance associated to the opened channel; this identifier will have to be used by all the subsequent script functions (**ETH_TCPS_$$$**) related to this communication server

The TCP server created with this function is able to handle up to **10** clients contemporarily.
Every time a new client connects to the server, a new dedicated communication socket is created for it.
The listening socket will keep on working for as long as needed, to accept all the supported clients connections.
Connection attempts from clients after this limit has been reached will result in failure.

This function affects the variable **ETH_NUMBER**, meant to count the number of open ethernet channels.
The server listening socket is counted only: the communication channels created later upon clients requests won't affect this variable (the created server counts as 1, regardless the number of clients connected to it).

The maximum number of ethernet channels managed by scripts is currently limited to **10**. Once the **ETH_NUMBER** reaches this limit, no more channels can be opened: new **ETH_$$$_OPEN** calls will fail.

ETH_TCPS_CLOSE

Closes an ethernet communication channel maintained as a TCP server.
Could be aimed to either the whole server domain, or to specific client connections.

ETH_TCPS_CLOSE (ETHID [, CLIENTIP, CLIENTPORT])

input			
ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if missing, the function is intended to be used to close the whole server domain: the server won't be listening for clients anymore, and all the existing clients connections are shut down; if given, the function is intended to be used to close only the connection with the specified client; the server will keep on listening for incoming connection requests
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

The targeted ETHID must be related to a TCP server.

Depending on the existence of a client address, the function can be used to affect a single client connection or the whole server with all its current connections.

This function affects the variable **ETH_NUMBER**, meant to count the number of open ethernet channels.

This function reports a successful result if the system is able to reset the channel and remove it from the managed ones; if, in the process, something goes wrong on the socket side, the variable **ETH_ERROR** might give details about the issue.

ETH_TCPS_CLIENTSNUMBER

Retrieves the number of clients currently connected to the TCP server.

NUMBER = **ETH_TCPS_CLIENTSNUMBER** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**

output		
NUMBER :	UDINT	the number of clients connected

The targeted ETHID must be related to a TCP server.

The returned value (within the range **0**..**10**) acts as limit for the functions that require a client's index among their parameters (see **ETH_TCPS_GETIPCLIENT**).

ETH_TCPS_GETIPLOCAL

Retrieves the IP address set up as local (server) address when the listening socket was created.

IP = **ETH_TCPS_GETIPLOCAL** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**

output		
IP :	UDINT	the (normalized) IP address of the local server; it's the local address that was explicitly given when the server was created by the function **ETH_TCPS_OPEN**

The targeted ETHID must be related to a TCP server.

As already mentioned, the returned IP address is in normalized (numeric) form.
Along with it, a couple of further variables are set up:
- **ETH_IPADDRESS** will replicate the returned IP address
- **ETH_IPPORT** will give the IP port associated with the communication channel

These variables are affected by every successful execution of this function; old values will be retained in case of errors. Note that several other ethernet functions are meant to affect them though (see their description in the <VARIABLES> section).

ETH_TCPS_GETIPCLIENT

Retrieves the IP address of a specific remote TCP client currently connected.

IP = **ETH_TCPS_GETIPCLIENT** (ETHID, INDEX)

input		
ETHID :	UDINT	the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
INDEX :	ANY_INT	the index (base-0) of the client among those connected to this server; the value of this parameter should be limited by the current value of **ETH_TCPS_CLIENTSNUMBER**
output		
IP :	UDINT	the (normalized) IP address of a remote client; it's the remote address recognized by the server when the client connection request was accepted

The targeted ETHID must be related to a TCP server.

As said, the index is supposed to be a number between 0 and **ETH_TCPS_CLIENTSNUMBER** (-1).
This means the range is affected by every client added and removed.
Clients additions happen automatically every time a client connection request is accepted by a listening server (the server will be listening until the maximum number of connected clients is reached). Clients removals happen only when explicitly requested by an **ETH_TCPS_CLOSE** call.
The clients will be maintained in chronological order, depending on their connection time. This means that new clients are always appended with the highest index, and also that when a client is removed (closed) all those with higher index will have the index adjusted (decreased) so that their count will remain continuous.
For example:
- clients A, B and C are added one after the other
  > their indexes will be: A(0), B(1), C(2)
- client B is closed
  > the indexes will become: A(0), C(1)

As already mentioned, the returned IP address is in normalized (numeric) form.
Along with it, a couple of further variables are set up:
- **ETH_IPADDRESS** will replicate the returned IP address
- **ETH_IPPORT** will give the IP port associated with the communication channel

These variables are affected by every successful execution of this function; old values will be retained in case of errors. Note that several other ethernet functions are meant to affect them though (see their description in the <VARIABLES> section).

ETH_TCPS_DATALENGTH

Retrieves the number of bytes currently available for reading from a given channel.

LENGTH = **ETH_TCPS_DATALENGTH** (ETHID [, CLIENTIP, CLIENTPORT])

input			
ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if given, the function is used to access only the specified client; if missing, the function is used to check all the connected clients and access information of whichever one is first found to have available data
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

output		
LENGTH :	UDINT	the number of available bytes, already readable from the given channel

The targeted ETHID must be related to a TCP server.

If the client IP address and port are explicitly given as parameters, then the identified client must be among the connected ones. In this case the identification parameters are replicated in the variables:
- **ETH_IPADDRESS** a client IP address
- **ETH_IPPORT** a client IP port

and the server retrieves information about that client only.
If the client identification parameters are not given instead, then this function sets the variables above with information related to the first client found to be useful:
1. if the server finds among all the connected clients at least one with available data, then the size of that client data is returned, and its address and port are set up in these variables;
3. if the server finds no data from any of the connected clients, then the returned value is 0 and these variables are set to 0.

In case of execution errors, these variables are not changed.

Note that even in case of successful execution, if information inconsistencies are found on the socket side, error codes might be notified by the **ETH_ERROR** variable.

ETH_TCPS_READBYTE

Reads a single byte of data from a given ethernet channel.

VALUE = **ETH_TCPS_READBYTE** (ETHID, TIMEOUT [, CLIENTIP, CLIENTPORT])

input

ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
TIMEOUT :	ANY_INT		a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if given, the function is used to access only the specified client; if missing, the function is used to check all the connected clients and read data from whichever one is first found to have some available
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

output

VALUE :	BYTE	the value of the acquired byte

The targeted ETHID must be related to a TCP server.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired.
Being this function meant to read 1 only byte, the variable value will be 1 in case data was available for reading, or 0 otherwise.
In case of errors, the variable remains unaffected.

If the client IP address and port are explicitly given as parameters, then the identified client must be among the connected ones. In this case the identification parameters are replicated in the variables:
- **ETH_IPADDRESS** a client IP address
- **ETH_IPPORT** a client IP port

and the server reads data from that client only.
If the client identification parameters are not given instead, then this function sets the variables above with information related to the first client found to be useful:
1. if the server finds among all the connected clients at least one with available data, then data is acquired from its channel, and its address and port are set up in these variables;
3. if the server finds no data from any of the connected clients, then nothing can be read, and these variables are set to 0 (along with the **ETH_RXLENGTH**).

In case of execution errors, these variables are not changed.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_TCPS_READBUFFER

Reads a whole buffer of data from a given ethernet channel.

VALUE = **ETH_TCPS_READBUFFER** (ETHID, SIZE, TIMEOUT [, CLIENTIP, CLIENTPORT])

input

ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
SIZE :	ANY_INT		the maximum number of bytes to get from the ethernet; this value is currently limited to **65536** bytes (64KB)
TIMEOUT :	ANY_INT		a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if given, the function is used to access only the specified client; if missing, the function is used to check all the connected clients and read data from whichever one is first found to have some available
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

output

VALUE :	ANY	the value of the acquired buffer; despite the generic declaration, this result is always an array of bytes; the size of the array is defined by the input parameter SIZE

The targeted ETHID must be related to a TCP server.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired.
The stored value could be anything between 0 and SIZE, depending on the amount of data available for reading.
In case of errors, the variable remains unaffected.

Upon identification of a target client, this function affects the variables **ETH_IPADDRESS** and **ETH_IPPORT** (see variables and **ETH_TCPS_READBYTE** for details).

The given SIZE is the number of elements of the returned array, and the maximum number of bytes acquired from the ethernet. The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned array is completely reset, and the variable **ETH_RXLENGTH** is set to 0;
- if any number of bytes within the given SIZE are found to be available, then all these bytes are acquired and returned, the remaining part of the array remains reset, and the variable **ETH_RXLENGTH** is set to that exact number of bytes;
- if more than SIZE bytes are available, then only the first SIZE ones are read and returned; the variable **ETH_RXLENGTH** is set to SIZE.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPS_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_TCPS_READSTRING

Reads a whole buffer of data in string (short) form from a given ethernet channel.

VALUE = **ETH_TCPS_READSTRING** (ETHID, SIZE, TIMEOUT [, CLIENTIP, CLIENTPORT])

input

ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
SIZE :	ANY_INT		the maximum number of characters to get from the ethernet; this value is currently limited to **65536** characters (64KB)
TIMEOUT :	ANY_INT		a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if given, the function is used to access only the specified client; if missing, the function is used to check all the connected clients and read data from whichever one is first found to have some available
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

output

VALUE :	STRING	the value of the acquired string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a TCP server.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired (same as the number of characters).
The stored value could be anything between 0 and SIZE, depending on the amount of data available for reading.
In case of errors, the variable remains unaffected.

Upon identification of a target client, this function affects the variables **ETH_IPADDRESS** and **ETH_IPPORT** (see variables and **ETH_TCPS_READBYTE** for details).

The given SIZE is the number of characters the returned string will be allocated for (and the maximum number of bytes acquired from the ethernet). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned.
See **ETH_TCPC_READSTRING** for details about the behaviour of a similar read function with respect to the management of the data acquisition, output preparation and **ETH_RXLENGTH** setting.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPC_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_TCPS_READWSTRING

Reads a whole buffer of data in string (wide) form from a given ethernet channel.

VALUE = **ETH_TCPS_READWSTRING** (ETHID, SIZE, TIMEOUT [, CLIENTIP, CLIENTPORT])

input

ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
SIZE :	ANY_INT		the maximum number of characters to get from the ethernet; this value is currently limited to **32768** characters (64KB)
TIMEOUT :	ANY_INT		a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0
CLIENTIP :	UDINT	[OPTIONAL]	the (normalized) IP address of a connected client; given only together with the CLIENTPORT; if given, the function is used to access only the specified client; if missing, the function is used to check all the connected clients and read data from whichever one is first found to have some available
CLIENTPORT :	ANY_INT	[OPTIONAL]	the IP port of a connected client; given only together with the CLIENTIP (see behaviour above)

output

VALUE :	WSTRING	the value of the acquired (wide) string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a TCP server.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired; note that this is NOT the same as the number of characters (with wide strings involved, each character is made of 2 bytes).
The stored value could be anything between 0 and SIZE*2, depending on the amount of data available for reading, and is supposed to always be an even value (multiple of 2, being the characters made of 2 bytes each).
In case of errors, the variable remains unaffected.

Upon identification of a target client, this function affects the variables **ETH_IPADDRESS** and **ETH_IPPORT** (see variables and **ETH_TCPS_READBYTE** for details).

The given SIZE is the number of characters the returned string will be allocated for (the maximum number of bytes acquired from the ethernet will be twice as much). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned.
See **ETH_TCPC_READWSTRING** for details about the behaviour of a similar read function with respect to the management of the data acquisition, output preparation and **ETH_RXLENGTH** setting.

At any given time this function should be able to read as many bytes as stated by **ETH_TCPC_DATALENGTH**.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_TCPS_WRITE

Writes data on an opened ethernet channel.

ETH_TCPS_WRITE (ETHID, VALUE [, SIZE], CLIENTIP, CLIENTPORT)

input

ETHID :	UDINT		the identifier of an existing server channel; must be a valid identifier returned by a previous call to **ETH_TCPS_OPEN**
VALUE :	ANY		the value that has to be transmitted; this could be a value of any type; the transmitted data will be the exact binary image of the given value; the programmer is supposed to know the binary format of the used types
SIZE :	ANY_INT	[OPTIONAL]	size (in bytes) of the part of VALUE sent on the ethernet; can be used to limit the number of bytes actually transmitted in cases where the data is passed in buffers (likely arrays) that might either be full or not; if this parameter is missing, then the whole VALUE is written; giving a SIZE bigger than the passed VALUE has no effect: the transmission is limited to the actual VALUE size anyway
CLIENTIP :	UDINT		the IP address (normalized) of the target client; together with the CLIENTPORT is used to identify the client (among the connected ones) to whom the data is sent
CLIENTPORT :	ANY_INT		the IP port of the target client; used together with the CLIENTIP; see above

The targeted ETHID must be related to a TCP server.

The function transmits exactly the number of bytes required by the type of the given value.
Any type of value can be passed to the function, included arrays, structures and strings.
The programmer must be aware of the exact format, size and endianity of the written data.
Note that even strings follow the same rule: plain STRINGs are sent with 1 byte per character, while WSTRINGs are sent using 2 bytes per character.
A degree of management on plain strings is implemented actually: whereas arrays of strings are sent like binary chunks of data (one byte for each byte of data buffer), plain strings are in fact terminated at their NUL terminator.

The variable **ETH_TXLENGTH** will count the number of bytes transmitted by a successful execution of this function (failed executions will leave the variable unchanged).

In case of successful result, if the function couldn't write anything (**ETH_TXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_UDP_OPEN

Open an ethernet communication channel in (connection-less) UDP mode.

ETHID = **ETH_UDP_OPEN** (IP, PORT)

input

IP :	UDINT	the (normalized) IP address for the (local) socket; identifies the address of the ethernet port used by the channel to transmit and receive data; this parameter could be given as ETHADDRESSANY to specify a channel that is not bound to a specific address: the channel will be able to receive messages from any ethernet port
PORT :	ANY_INT	the IP port opened by the (local) socket; this is a mandatory specification (explicitly forbidden to be an ETHPORTAUTO): it's the port that has to be used by remote partners to reach this channel

output

ETHID :	UDINT	the identifier of the created UDP channel; it is defined as the index (base-0) of the ethernet manager instance associated to the opened channel; this identifier will have to be used by all the subsequent script functions (ETH_UDP_$$$) related to this communication channel

This function affects the variable ETH_NUMBER, meant to count the number of open ethernet channels.

The maximum number of ethernet channels managed by scripts is currently limited to **10**. Once the ETH_NUMBER reaches this limit, no more channels can be opened: new ETH_$$$_OPEN calls will fail.

ETH_UDP_CLOSE

Closes an ethernet communication channel previously opened in UDP mode.

ETH_UDP_CLOSE (ETHID)

input		
ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**

The targeted ETHID must be related to a UDP channel.

This function affects the variable **ETH_NUMBER**, meant to count the number of open ethernet channels.

This function reports a successful result if the system is able to reset the channel and remove it from the managed ones; if, in the process, something goes wrong on the socket side, the variable **ETH_ERROR** might give details about the issue.

ETH_UDP_GETIPLOCAL

Retrieves the local IP address set up when the channel was created.

IP = **ETH_UDP_GETIPLOCAL** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**

output		
IP :	UDINT	the (normalized) IP address of the local socket; it's the local address explicitly given when the UDP channel was created by the function **ETH_UDP_OPEN**

The targeted ETHID must be related to a UDP channel.

As already mentioned, the returned IP address is in normalized (numeric) form.
Along with it, a couple of further variables are set up:
- **ETH_IPADDRESS** will replicate the returned IP address
- **ETH_IPPORT** will give the IP port associated with the communication channel

These variables are affected by every successful execution of this function; old values will be retained in case of errors. Note that several other ethernet functions are meant to affect them though (see their description in the <VARIABLES> section).

ETH_UDP_DATALENGTH

Retrieves the number of bytes currently available for reading from a given channel.

LENGTH = **ETH_UDP_DATALENGTH** (ETHID)

input		
ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
output		
LENGTH :	UDINT	the number of available bytes, already readable from the given channel

The targeted ETHID must be related to a UDP channel.

This function will return the total number of bytes waiting to be read.
There is no way to retrieve information about the number of datagrams, nor about the amount of data received from specific remote partners.

Note that even in case of successful execution, if information inconsistencies are found on the socket side, error codes might be notified by the **ETH_ERROR** variable.

NOTE: this function behaves in slightly different ways on different platforms:
- Windows: the function returns the total number of bytes currently readable from the UDP channel (the sum of the lengths of all the pending datagrams);
- Linux: the function only returns the length of the next pending datagram.

ETH_UDP_READBYTE

Reads a single byte of data from a given ethernet channel.

VALUE = **ETH_UDP_READBYTE** (ETHID, TIMEOUT)

input		
ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output		
VALUE :	BYTE	the value of the acquired byte

The targeted ETHID must be related to a UDP channel.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired.
Being this function meant to read 1 only byte, the variable value will be 1 in case data was available for reading, or 0 otherwise.
In case of errors, the variable remains unaffected.

Remember that UDP communication is based on whole datagrams, not on streams of bytes: if an **ETH_UDP_READBYTE** instruction is used to read from a larger datagram, then the 1st byte of the packet is returned, and the remaining part is simply lost.

Upon successful execution, this function is able to set the variables
- **ETH_IPADDRESS** a partner IP address
- **ETH_IPPORT** a partner IP port

with the information related to the partner that actually transmitted the acquired data.
In case of execution errors, these variables are not changed.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_UDP_READBUFFER

Reads a whole buffer of data from a given ethernet channel.

VALUE = **ETH_UDP_READBUFFER** (ETHID, SIZE, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
SIZE :	ANY_INT	the maximum number of bytes to get from the ethernet; this value is currently limited to **65536** bytes (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	ANY	the value of the acquired buffer; despite the generic declaration, this result is always an array of bytes; the size of the array is defined by the input parameter SIZE

The targeted ETHID must be related to a UDP channel.

In case of successful execution, the function returns an array of bytes with the given SIZE. Not necessarily all the bytes of the array have to be filled up with read bytes: some might be reset to 0.
The variable **ETH_RXLENGTH** will count the number of bytes read and returned.

The given SIZE is the number of elements of the returned array, and the maximum number of bytes acquired from the ethernet. The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned:
- if nothing is found to be available within the given TIMEOUT, then nothing is read, the returned array is completely reset, and the variable **ETH_RXLENGTH** is set to 0;
- if any number of bytes within the given SIZE are found to be available, then all these bytes are acquired and returned, the remaining part of the array remains reset, and the variable **ETH_RXLENGTH** is set to that exact number of bytes;
- if more than SIZE bytes are available, then only the first SIZE ones are read and returned; the variable **ETH_RXLENGTH** is set to SIZE.

At any given time this function should be able to read as many bytes as stated by **ETH_UDP_DATALENGTH**.

Remember that UDP communication is based on whole datagrams, not on streams of bytes: if an **ETH_UDP_READBUFFER** with a given size is used to read from a larger datagram, then the initial bytes of the packet are returned, and the remaining part is simply lost.

Upon successful execution, this function is able to set the variables
- **ETH_IPADDRESS** a partner IP address
- **ETH_IPPORT** a partner IP port

with the information related to the partner that actually transmitted the acquired data.
In case of execution errors, these variables are not changed.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

ETH_UDP_READSTRING

Reads a whole buffer of data in string (short) form from a given ethernet channel.

VALUE = **ETH_UDP_READSTRING** (ETHID, SIZE, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
SIZE :	ANY_INT	the maximum number of characters to get from the ethernet; this value is currently limited to **65536** characters (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	STRING	the value of the acquired string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a UDP channel.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired (same as the number of characters).
The stored value could be anything between 0 and SIZE, depending on the amount of data available for reading.
In case of errors, the variable remains unaffected.

The given SIZE is the number of characters the returned string will be allocated for (and the maximum number of bytes acquired from the ethernet). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned.
See **ETH_TCPC_READSTRING** for details about the behaviour of a similar read function with respect to the management of the data acquisition, output preparation and **ETH_RXLENGTH** setting.

At any given time this function should be able to read as many bytes as stated by **ETH_UDP_DATALENGTH**.

Remember that UDP communication is based on whole datagrams, not on streams of bytes: if an **ETH_UDP_READSTRING** with a given size is used to read from a larger datagram, then the initial characters of the packet are returned, and the remaining part is simply lost.

Upon successful execution, this function is able to set the variables
- **ETH_IPADDRESS** a partner IP address
- **ETH_IPPORT** a partner IP port

with the information related to the partner that actually transmitted the acquired data.
In case of execution errors, these variables are not changed.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong. Similarly, in case issues arise on the socket side after a part of string has been acquired already, then the function successfully returns what has been read in the output string, and let the variable **ETH_ERROR** give details about the error.

ETH_UDP_READWSTRING

Reads a whole buffer of data in string (wide) form from a given ethernet channel.

VALUE = **ETH_UDP_READWSTRING** (ETHID, SIZE, TIMEOUT)

input

ETHID :	UDINT	the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
SIZE :	ANY_INT	the maximum number of characters to get from the ethernet; this value is currently limited to **32768** characters (64KB)
TIMEOUT :	ANY_INT	a read timeout: the number of milliseconds the function is allowed to wait in case nothing is available for reading yet; can be 0

output

VALUE :	WSTRING	the value of the acquired (wide) string; the allocated size of the string is defined by the input parameter SIZE; the actual length of the string content though depends on the data available for reading

The targeted ETHID must be related to a UDP channel.

After a successful execution, in the variable **ETH_RXLENGTH** is stored the number of bytes acquired; note that this is NOT the same as the number of characters (with wide strings involved, each character is made of 2 bytes).
The stored value could be anything between 0 and SIZE*2, depending on the amount of data available for reading, and is supposed to always be an even value (multiple of 2, being the characters made of 2 bytes each).
In case of errors, the variable remains unaffected.

The given SIZE is the number of characters the returned string will be allocated for (the maximum number of bytes acquired from the ethernet will be twice as much). The function waits for something to be available for reading from the port; as soon as something is available, it is acquired and returned.
See **ETH_TCPC_READWSTRING** for details about the behaviour of a similar read function with respect to the management of the data acquisition, output preparation and **ETH_RXLENGTH** setting.

At any given time this function should be able to read as many bytes as stated by **ETH_UDP_DATALENGTH**.

Remember that UDP communication is based on whole datagrams, not on streams of bytes: if an **ETH_UDP_READWSTRING** with a given size is used to read from a larger datagram, then the initial characters of the packet are returned, and the remaining part is simply lost.

Upon successful execution, this function is able to set the variables
- **ETH_IPADDRESS** a partner IP address
- **ETH_IPPORT** a partner IP port

with the information related to the partner that actually transmitted the acquired data.
In case of execution errors, these variables are not changed.

In case of successful result, if the function couldn't read anything (**ETH_RXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong. Similarly, in case issues arise on the socket side after a part of string has been acquired already, then the function successfully returns what has been read in the output string, and let the variable **ETH_ERROR** give details about the error.

ETH_UDP_WRITE

Writes data on an opened ethernet channel.

ETH_UDP_WRITE (ETHID, VALUE [, SIZE], DESTIP, DESTPORT)

input

ETHID :	UDINT		the identifier of an existing UDP channel; must be a valid identifier returned by a previous call to **ETH_UDP_OPEN**
VALUE :	ANY		the value that has to be transmitted; this could be a value of any type; the transmitted data will be the exact binary image of the given value; the programmer is supposed to know the binary format of the used types
SIZE :	ANY_INT	[OPTIONAL]	size (in bytes) of the part of VALUE sent on the ethernet; can be used to limit the number of bytes actually transmitted in cases where the data is passed in buffers (likely arrays) that might either be full or not; if this parameter is missing, then the whole VALUE is written; giving a SIZE bigger than the passed VALUE has no effect: the transmission is limited to the actual VALUE size anyway
DESTIP :	UDINT		the IP address (normalized) of the UDP partner; together with the DESTPORT is used to identify the target
DESTPORT :	ANY_INT		the IP port of the UDP partner; used together with the DESTIP; see above

The targeted ETHID must be related to a UDP channel.

The function transmits exactly the number of bytes required by the type of the given value.
Any type of value can be passed to the function, included arrays, structures and strings.
The programmer must be aware of the exact format, size and endianity of the written data.
Note that even strings follow the same rule: plain STRINGs are sent with 1 byte per character, while WSTRINGs are sent using 2 bytes per character.
A degree of management on plain strings is implemented actually: whereas arrays of strings are sent like binary chunks of data (one byte for each byte of data buffer), plain strings are in fact terminated at their NUL terminator.

The variable **ETH_TXLENGTH** will count the number of bytes transmitted by a successful execution of this function (failed executions will leave the variable unchanged).

In case of successful result, if the function couldn't write anything (**ETH_TXLENGTH** reports 0 bytes length) the variable **ETH_ERROR** might give further details about what could have gone wrong.

< VARIABLES >

The following are the variables usable to share information and directives for the ethernet channels management:

ETH_ERROR
type ULINT
access R
variable set with additional result information by several ETH_$$$ functions;
0 indicates a success; anything else is an error code;
this variable is updated by successful executions of the functions: ETH_PING, ETH_$$$_CLOSE, ETH_$$$_DATALENGTH, ETH_$$$_READ$$$, ETH_$$$_WRITE;
it's used to report issues in case the function execution has been limited by errors on the socket side; it's implemented to allow the programmer to work with ethernet functions without the need to systematically disable blocking errors, to face issues coming from remote connections

ETH_NUMBER
type UDINT
access R
gives the number of ethernet communication channels currently opened by scripts

ETH_RXLENGTH
type UDINT
access R
gives the number of bytes acquired by the last successful execution of a read function (see ETH_$$$_READBYTE, ETH_$$$_READBUFFER, ETH_$$$_READSTRING, and ETH_$$$_READWSTRING);
failed calls won't affect this variable;
the given value could be 0 in case the invoked function found nothing to read from the ethernet channel

ETH_TXLENGTH
type UDINT
access R
gives the number of bytes transmitted by the last successful execution of a write function (see ETH_$$$_WRITE);
failed calls won't affect this variable

ETH_IPADDRESS
type UDINT
access R
gives the normalized IP address of a socket end-point: several functions use it to add extended information to their returned data (normally given together with ETH_IPPORT)
(see ETH_$$$_GETIPLOCAL, ETH_TCPC_GETIPSERVER, ETH_TCPS_GETIPCLIENT, ETH_TCPS_DATALENGTH, ETH_TCPS_READ$$$, ETH_UDP_READ$$$);
see ETH_IP and ETH_GETIP for information about normalized addresses and ways to convert them

ETH_IPPORT
type UINT
access R
gives the IP port number of a socket end-point: several functions use it to add extended information to their returned data (normally given together with ETH_IPADDRESS)
(as above, see ETH_$$$_GETIPLOCAL, ETH_TCPC_GETIPSERVER, ETH_TCPS_GETIPCLIENT, ETH_TCPS_DATALENGTH, ETH_TCPS_READ$$$, ETH_UDP_READ$$$)

ETH_IP1
type UDINT

ETH_IP2 type UDINT
ETH_IP3 type UDINT
ETH_IP4 type UDINT
access R

the variables give the 4 components of the IP address converted by the function **ETH_GETIP**

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
ETHADDRESSANY	0	**ETH_TCPS_OPEN**, **ETH_UDP_OPEN**
ETHPORTAUTO	0	**ETH_TCPC_OPEN**

8. COMMON - LIBRARIES

8.1. Standard libraries

Standard libraries management is implemented for generic Windows DLLs and Linux Shared Objects.
The following are the available methods, all based on the assumption that the programmer knows exactly how the library functions are defined, and is able to transport their interface with binary compatibility using ST data types.
Libraries whose data binary configuration won't match that of the available ST types will not be usable in the script.
Detailed instruction on how to match parameters and return data types are given below.

LIBRARY_LOAD

Loads a dynamic library.

LIBID = **LIBRARY_LOAD** (NAME)

input

NAME :	ANY_STRING	path and name of the library file

output

LIBID :	UDINT	a unique numeric identifier of the loaded library; this ID will have to be used in future calls to functions like LIBRARY_RELEASE and LIBRARY_FXLOAD

External dynamic libraries can be loaded and used by the runtime (libraries such as Windows DLL or Linux Shared Objects), provided:
- only plain functions are referenced (no classes, variables or any other kind of shared symbol),
- only ST compatible types are used for functions parameters and returns (see LIBRARY_FXLOAD).

example

```
ST_OPTION HANDLE_ERRORS;

VAR
   libid : UDINT;
   fxid  : UDINT;
   ret_i : DINT;
   ret_b : ARRAY [100] OF USINT;
END_VAR;

libid := LIBRARY_LOAD ('/home/my/path/libtest.so');
IF FXRESULT = 0 THEN
   // fxid := LIBRARY_FXLOAD (libid, "FxVoid");        // defining a void function
   // fxid := LIBRARY_FXLOAD (libid, "FxInt", ret_i);  // defining an 'int' function
   fxid := LIBRARY_FXLOAD (libid, "FxArray", ret_b);   // a function returning an array
   IF FXRESULT = 0 THEN
      LIBRARY_FXCALL (fxid, ANY_TO_DINT(123), 'STR');
      // LIBRARY_FXRELEASE (fxid);  // can be skipped since the <LIBRARY_RELEASE> already follows
   ELSE
      // handle load error
```

```
   END_IF;
   LIBRARY_RELEASE (libid);
ELSE
   // handle load error
END_IF;
```

LIBRARY_RELEASE

Releases a previously loaded dynamic library.

LIBRARY_RELEASE (LIBID)

input

LIBID :	UDINT	the numeric identifier of a library previously loaded by a **LIBRARY_LOAD**

The function releases the library itself along with all the resources allocated with it.

If still in use (still not released) all the loaded library's functions are released as well: this makes explicit calls to **LIBRARY_FXRELEASE** completely optional (the function can still be used to make mechanics cleaner, or to handle cases where too many different functions of too many libraries have to be contemporarily used, since there is a limit to the number of library functions available at any given time).

See the *example* given with the **LIBRARY_LOAD** specifications.

LIBRARY_FXLOAD

Obtains a reference to a function exported by a dynamic library.

FXID = **LIBRARY_FXLOAD** (LIBID, NAME [, RETURN])

input

LIBID :	UDINT		the numeric identifier of a library previously loaded by a **LIBRARY_LOAD**
NAME :	ANY_STRING		name of the function that has to be retrieved from the LIBID library
RETURN :	ANY	[OPTIONAL]	this parameter is used to declare the type of the value returned by the specified function; it could be a variable or a value of well-defined type, but the given value itself is not important: the only thing that matters about this parameter is the type; when later invoked by the **LIBRARY_FXCALL**, this is the type used to build its return value; if missing, the given function is expected to have no return (expected to be <void>); it is important to declare the most appropriate type for the specified function, not only to obtain a proper output value, but also to be able to keep the best binary compatibility with the external library; **errors in this declaration might lead to runtime crashes when the referenced library function is invoked**

output

FXID :	UDINT	a unique numeric identifier of the given function; this ID will have to be used in future calls to functions like **LIBRARY_FXRELEASE** and **LIBRARY_FXCALL**

There is a limit to the maximum number of functions that can be loaded (currently set to **128**).
If more functions are needed, then programmers might need to release some (**LIBRARY_FXRELEASE**) before to load others.

In case of functions that return a value, the declared return type is required to have binary compatibility with the actual returned value. In particular, types that can/should be used are:

SINT	char
INT	short int
DINT	long int
LINT	long long int
USINT | BYTE	unsigned char
UINT | WORD	unsigned short int
UDINT | DWORD	unsigned long int
LUINT | LWORD	unsigned long long int
REAL	float
LREAL	double

along with:

<ranges>	the actual type is their base type
<enumeratives>	considered to be UDINT (unsigned 32 bits)
<strings>	of both ansi and wide type (STRING, WSTRING)
<arrays>	of whatever dimension/s of the types above
<structures>	containing fields of any of the types above

(structures of arrays and arrays of structures are welcome as well).
Mainly the only thing that matters is that the binary image of the returned value must match the binary image of the declared type.

A limitation: POINTERS are NOT allowed (yet, since pointers are not supported yet by the ST engine).
Side effect of the matter is that, even though strings and arrays can be handled when directly returned by the invoked functions (both strings and arrays are returned by library functions as pointers to their value memory but can be implicitly converted in the pointed value content), when they are part of structures, they can't be exchanged in pointer form, since that would require pointer management in the ST language itself. They should be contained in their structure in full form instead.
See the following examples:

example

```
// "C" prototype of library function:
// char * StringFx ();

VAR
   ret_s : STRING [10];
END_VAR;

fxid := LIBRARY_FXLOAD (libid, "StringFx", ret_s);  // This is ALLOWED
ret_s := LIBRARY_FXCALL (fxid);                     // plain strings are returned as pointer by the library
                                                    // and are then implicitly converted by ST
// "C" prototype of library function:
// typedef struct mystr { char mystring[10+1]; } STR;
// STR StringFx ();

TYPE STR :
   STRUCT
      fld_s : STRING [10];
   END_STRUCT;
END_TYPE;
VAR
   ret_s : STR;
END_VAR;

fxid := LIBRARY_FXLOAD (libid, "StringFx", ret_s);  // This is ALLOWED
ret_s := LIBRARY_FXCALL (fxid);                     // strings inside structures are returned in full form
                                                    // by the library; the plain binary copy of the
                                                    // structure content then works fine in ST
// "C" prototype of library function:
// typedef struct mystr { char * mystring; } STR;
// STR StringFx ();

TYPE STR :
   STRUCT
      fld_s : STRING [10];
   END_STRUCT;
END_TYPE;
VAR
   ret_s : STR;
END_VAR;

fxid := LIBRARY_FXLOAD (libid, "StringFx", ret_s);  // This is FORBIDDEN
ret_s := LIBRARY_FXCALL (fxid);                     // strings held in structures in pointer form are
                                                    // copied as such and can't be handled in ST
```

This last case is forbidden because the binary image of the "C" structure and that of the "ST" structure are different. The same goes for arrays and for any kind of bufferized value passed in pointer form.

Another limitation: the total size of the returned value must not exceed a limit currently set to **1024** bytes.

See the *example* given with the **LIBRARY_LOAD** specifications.

LIBRARY_FXRELEASE

Releases a previously loaded library function.

LIBRARY_FXRELEASE (FXID)

input

FXID :	UDINT	the numeric identifier of a function previously loaded by a **LIBRARY_FXLOAD**

The function releases the function ID only.
Keep in mind that, after a release, the given ID might be reused by future **LIBRARY_FXLOAD**.

See the *example* given with the **LIBRARY_LOAD** specifications.

LIBRARY_FXCALL

Executes the function of a dynamic library.

[RETURN =] **LIBRARY_FXCALL** (FXID [, PARAM1 [, PARAM2 [, ...]]])

input			
FXID :	UDINT		the numeric identifier of a library function previously loaded by a **LIBRARY_FXLOAD**
PARAM# :	ANY	[OPTIONAL]	these parameters are used to give the values that have to be passed as parameters to the invoked function; their number must match that of the actual parameters of the invoked function (no parameters at all is an allowed case as well of course); as in the case of the function return type, it is important to use values of the most appropriate types, in order to be able to keep the best binary compatibility with the external library; **errors in this list declaration might lead to runtime crashes due to incompatibility with the invoked function**
output			
RETURN :	ANY	[OPTIONAL]	this is the value returned by the invoked external function; formally the value could be of any type; at runtime the actual type will be the one that was declared when the function was "loaded" by the **LIBRARY_FXLOAD**; note that if the invoked function was declared to have no return, then this **LIBRARY_FXCALL** will have no return as well

If parameters have to be passed to the invoked function, then it is important to keep the best binary compatibility with the external library: number, type, size, endianity… everything about the passed parameters should match the expected ones.
See **LIBRARY_FXLOAD** for notes about the types that can/should be used (for parameters as well as for returned values). Types and limitations are the same.

A couple of more notes:
- being pointers not supported yet, output parameters can't be defined as well;
- as in the **LIBRARY_FXLOAD** return case, parameters found to be STRINGs or ARRAYs are implicitly automatically converted and passed as pointers to the library, so at least this kind of content can be transferred; structures instead are always exchanged by value;
- even the total size of the passed parameters (those directly fit in stack, excluded the pointed content) can't exceed a limit, currently set to **1024** bytes.

See the *example* given with the **LIBRARY_LOAD** and the **LIBRARY_FXLOAD** specifications.

8.2. COM libraries

With the intent to support - at least in part - the functionalities of our current scripted projects, written in VBScript, a family of functions is provided here for the management of COM objects registered in the system.
This implementation is limited to distributions for Windows PCs, since COM technology won't make sense in a Linux environment.
The following are the available methods, all structured trying to mitigate the issues coming from the fundamental incompatibility between the COM Variant types (natively part of the VBScript environment) and the strict data types management of ST.
Most of the existing COM objects should be supported by the current implementation; some rare data type match could be not handled yet, and might be considered for future improvements.

Detailed instruction on how to match parameters and returned data types are given below.
It is important to check out the whole 'Variant (COMVAR)' section introduction paragraphs, where most of the data type-related information are gathered.

8.2.1. Libraries usage model (COMLIB)

Using COM objects in ST is not a trivial task: several steps should be followed in order to properly load and use the involved objects, functions and data types.
The functions available in ST allow to:
- create and destroy the COM objects (see **COMLIB_LOAD** and **COMLIB_RELEASE**)
- load and release functions and properties made public by the COM (see **COMLIB_FXLOAD** and **COMLIB_FXRELEASE**)
- invoke the execution of the loaded COM functions (see **COMLIB_FXCALL**)
- access the values of the loaded COM properties (see **COMLIB_PROPGET** and **COMLIB_PROPSET**)

In short, the steps needed are the following. The process itself is pretty straightforward, but the details in the datatypes management can become an issue sometimes:
- create the COM object using the ST **COMLIB_LOAD**;
  the name given to the function must be the one registered in the system for the desired object;
  save the returned ID since it will be needed in all subsequent calls involving this COM object;
- load and declare the functions (and properties) needed from the COM object using the ST **COMLIB_FXLOAD**;
  the given function name must correspond to the one published by the COM;
  the declared in/out data types must be the closest match possible for COM and ST; precise rules are given in the COMVAR paragraphs below;
- the loaded functions and properties can now be used invoking the ST **COMLIB_FXCALL**, **COMLIB_PROPGET** and **COMLIB_PROPSET**;
  make sure to invoke the functions using the correct data types, as needed and previously declared;
  COMVAR functions (**COMVAR_$$$**) might be needed in case specific unsupported types have to be exchanged with the COM; also COMVAR functions can provide support for the COM arrays management; again, see details in the COMVAR paragraphs below;
- release the COM functions (and properties) using the ST **COMLIB_RELEASE** when no longer needed;
- release the COM object using the ST **COMLIB_RELEASE** when no longer needed.

8.2.2. Variants (COMVAR)

The "Variant" is the basic generic data type upon which all the implementation of the COM interfaces is based. Transporting values between ST and COM unmatching environments, always means that the system must be able to transform ST values in Variants, and Variants in ST values. Since Variants could encapsulate data types not natively recognizable by the ST engine, some additional management is required in some cases.

This additional management is not always needed:
sometimes ST values can be directly transformed in Variants, to be freely passed to the COM interface;
and sometimes Variants received from the COM interface can be directly transformed in ST values.
But it's also possible to face situations where the needed Variants are too complex or simply not compatible with ST native types.

In our ST environment we define an object, namely a "COMVAR", able to carry out the functionalities of the Variants (they effectively encapsulate Variants themselves) and fill up the gaps between ST and COM when it comes to transporting values between the two worlds.
COMVARs are identified by numeric IDs, used wherever they are referenced in an ST function; they can be freely created, destroyed, copied, read and written, provided the relation between their incapsulated Variant value and the corresponding ST value is clearly stated somewhere in the script.

In this section a family of functions (**COMVAR_$$$**) is specified for this kind of treatment, sometimes needed to prepare parameters and to examine results exchanged through COM functions.
Also important notes are given in the following paragraphs, regarding general rules and management of these objects: these rules are fundamental in the management of both COMLIB functions and COMVAR objects, and will be often referenced in this whole chapter.

8.2.2.1. Types declaration

Whenever a value has to be exchanged between a COMVAR and an ST variable, the better types match will have to be explicitly declared. This is something that happens often:
- every time a value is written in a COMVAR or in one of its elements (in case of arrays),
- every time a value is read from a COMVAR or one of its elements,
- every time a parameter is passed to a COM function (or property),
- every time a return value is obtained from a COM function (or property).

All the ST native elementary types can be referenced, along with their array form, and can be freely used in these exchanges with the COM Variants. Structured types instead are not supported by Variants and can't be used in COM exchanges.
On the Variants side, a limited subset made of the main existing types is supported; the following are the usable types (the data types the script is able to map from or to):
- **VT_I1**, **VT_I2**, **VT_I4**, **VT_I8**, **VT_UI1**, **VT_UI2**, **VT_UI4**, **VT_UI8** (the main integer types),
- **VT_R4**, **VT_R8** (the main floating-point types),
- **VT_INT**, **VT_UINT**, **VT_DECIMAL** (more integers interpreted in reads),
- **VT_BOOL** (boolean),
- **VT_BSTR** (all strings follow the OLESTR specification),
- **VT_DATE** (date and time information),
- **VT_DISPATCH** (the reference to a COM object interface).

Arrays (**VT_ARRAY**) are allowed as well. In case of arrays:
- multiple dimensions are allowed;
- they are all expected to be arrays of Variants (VT_ARRAY | VT_VARIANT);
- each element is a Variant itself and can have any of the supported types (further arrays excluded);
- strange forms of values layout, such as "byref" values, or encapsulations like "Variants of Variants of array of Variants" and the likes, are not allowed; in case of needs, specific extensions to the allowed formats might be considered in the future.

When types are specified in the script, the following codes can be used (see them in the **<CONSTANTS>** section as well):

ST symbolic	Notes
TYPESINT	| All the elementary ST types have a dedicated code that can be
TYPEINT	| used when a precise match is possible between ST and COM:
TYPEDINT	| first of all, all the plain numeric integer and floating-point values
TYPELINT	| have their obvious match
TYPEUSINT	|
TYPEUINT	|
TYPEUDINT	|
TYPEULINT	|
TYPEREAL	|
TYPELREAL	|
TYPETIME	| these types can be used as 'pseudo-numeric' values;
TYPELTIME	| the indication usually implies the conversion in unsigned integer values
TYPECHAR	| of matching size
TYPEWCHAR	| (for example, reading or writing a WCHAR is the same ad reading or writing a WORD,
TYPEBYTE	| which is the same as reading or writing a UINT: they all map to a **VT_UI2** Variant type)
TYPEWORD	|
TYPEDWORD	|
TYPELWORD	|
TYPEBOOL	used to define precise casts between the ST type and a Variant **VT_BOOL**
TYPESTRING	used to define precise casts between the ST type and a Variant **VT_BSTR**
TYPEWSTRING	used to define precise casts between the ST type and a Variant **VT_BSTR**
TYPEDATE	| date/time types are used in reads to automatically convert **VT_DATE**

TYPETOD	| Variants in specific ST formats
TYPELTOD	| (not formally usable in writes, where a generic **TYPEVDATE** is expected; see below)
TYPEDT	|
TYPELDT	|
TYPEVDATE	used with Variants containing a date (**VT_DATE**, with date and time information)
TYPEVOBJECT	used with Variants containing the reference to a COM object (**VT_DISPATCH**); a Variant like this carries exactly the pointer to the "Dispatch" interface of the COM; its value is transported in ST as a plain unsigned 32 bits integer (the numeric value of the pointer itself)
TYPECOMVAR	used with Variants containing values that are too complex to be directly translated in ST; it's the case, for example, of arrays made of elements of heterogeneous types; can be used also to mark Variants that won't need to be used in ST but will have to travel simply from COM to COM; in this case the "real" value of a Variant is simply stored in a COMVAR (not directly assigned to an ST variable); in place of this complex value, the ID of the COMVAR is assigned instead
TYPEARRAY	used in combination with other types, to state that the actual value is an array of that type; never to be used with **TYPECOMVAR** (in which case an array could be contained in a single COMVAR)
TYPEAUTO	used to declare Variants with which automatic values casts can be applied by the system; in these cases the script engine will try to use the best match possible for the given values; this declaration is often, but not necessarily always, enough
TYPENONE	in some cases used to mark values that should be excluded from the system computations; see relevant functions below

8.2.2.2. Writing values (from ST to COM)

This is something that happens every time a parameter is passed to a COM function (or assigned to a COM property), or every time a value is explicitly assigned to a COMVAR (or to an element of a COMVAR array).
When COM functions are involved, the parameter type is declared with the function **COMLIB_FXLOAD**; when COMVARs assignments are involved instead, the value type is directly stated with the functions **COMVAR_SET** and **COMVAR_SETELEMENT** (for all functions see specifications below).
In all cases, the declared type has to be intended as the desired type of the destination Variant, and will be used as an indication of how to transform the given (well-defined) source ST values.
In particular, see the following:

ST symbolic	Notes
TYPESINT	settles the ST value in a **VT_I1** Variant (8 bits signed integer value)
TYPEINT	settles the ST value in a **VT_I2** Variant (16 bits signed integer value)
TYPEDINT	settles the ST value in a **VT_I4** Variant (32 bits signed integer value)
TYPELINT	settles the ST value in a **VT_I8** Variant (64 bits signed integer value)
TYPEUSINT	settles the ST value in a **VT_UI1** Variant (8 bits unsigned integer value)
TYPEUINT	settles the ST value in a **VT_UI2** Variant (16 bits unsigned integer value)
TYPEUDINT	settles the ST value in a **VT_UI4** Variant (32 bits unsigned integer value)
TYPEULINT	settles the ST value in a **VT_UI8** Variant (64 bits unsigned integer value)
TYPEREAL	settles the ST value in a **VT_R4** Variant (32 bits floating-point value)
TYPELREAL	settles the ST value in a **VT_R8** Variant (32 bits floating-point value)
TYPETIME	considered synonym of **TYPEUDINT**
TYPELTIME	considered synonym of **TYPEULINT**
TYPECHAR	considered synonym of **TYPEUSINT**
TYPEWCHAR	considered synonym of **TYPEUINT**
TYPEBYTE	considered synonym of **TYPEUSINT**
TYPEWORD	considered synonym of **TYPEUINT**
TYPEDWORD	considered synonym of **TYPEUDINT**
TYPELWORD	considered synonym of **TYPEULINT**
TYPEBOOL	settles the ST value in a **VT_BOOL** Variant
TYPESTRING **TYPEWSTRING**	| either the STRING or WSTRING declaration can be used to define a destination Variant string; | there is no difference between the two declarations, from any point of view they can | be considered synonyms: in any case the Variant will be a **VT_BSTR**, made of wide characters, | and in any case the acceptable source ST values are both STRINGs and WSTRINGs
TYPEDATE **TYPETOD** **TYPELTOD** **TYPEDT** **TYPELDT**	| these types are explicitly forbidden to be used in assignments to Variants, since there is | no clear correspondence between the ST formats and a Variant type; | when dates and times have to be assigned, use the **TYPEVDATE** declaration instead (see below): | the system will then be able to accommodate the given ST values in the most appropriate way |
TYPEVDATE	this declaration means that the destination Variant has to be of a **VT_DATE** type; the source ST value can be given in different forms; the allowed source types include DATE, DT, LDT, TOD, LTOD (the Variant **VT_DATE** is able to contain this kind of information, even though precision and range don't always match); also different numeric values could be given: only in case of LAX types, the system allows the assignments of plain numbers directly in the date field of the Variant (of course the programmer is supposed to know its exact numeric format, and how to handle it)
TYPEVOBJECT	this declaration means that the ST value is providing an unsigned 32 bits integer containing the reference to a COM object; the reference is actually the pointer to the "Dispatch" interface of the object; the destination Variant will directly assume it as its value, along with a **VT_DISPATCH** type
TYPECOMVAR	this declaration means that the ST value is actually providing the ID of a COMVAR, and that the value of the COMVAR itself is the one that has to be assigned to the destination Variant; this is something that makes sense in case of parameters passed to a COM function: if the needed parameter is too complex for an ST value to be passed directly, then it can be prepared in a COMVAR first, and then passed as a reference to it;

	makes a lot less sense in case of assignments to COMVARs; not forbidden anyway: could be seen as a way to copy values from a COMVAR to another (a **COMVAR_COPY** function exists anyway)
TYPEARRAY	used in combination with all the types above, except TYPECOMVAR and those explicitly forbidden
TYPEAUTO	used when the script is trusted to be able to automatically decide the best match between the given ST value and the destination COM Variant (the destination type is automatically decided); in this case: - numeric values are mapped on standard **VT_I1**, **VT_I2**, **VT_I4**, **VT_I8**, **VT_UI1**, **VT_UI2**, **VT_UI4**, **VT_UI8**, **VT_R4**, **VT_R8** - all bitstrings except booleans (BYTEs, WORDs, DWORDs, LWORDs) are similarly mapped on the numeric (unsigned integer) values of matching size - the same goes for CHARs, WCHARs, TIMEs and LTIMEs, used as numeric values and mapped on the matching unsigned integers - booleans are mapped on **VT_BOOL** - strings are mapped on **VT_BSTR** - the remaining date/time types (DATE, DT, LDT, TOD, LTOD) are mapped on **VT_DATE**
TYPENONE	NEVER used when ST to COM assignments happen

8.2.2.3. Reading values (from COM to ST)

This is something that happens every time a value is returned by a COM function (or retrieved from a COM property), or every time a value is explicitly read from a COMVAR (or from an element of a COMVAR array).
When COM functions are involved, the return type is declared with the function **COMLIB_FXLOAD**; when COMVARs readings are involved instead, the value type is directly stated with the functions **COMVAR_GET** and **COMVAR_GETELEMENT** (for all functions see specifications below).
In all cases, the declared type has to be intended as the exact type formally needed by the destination ST value, and will be used as an indication of how to create it starting from the given (well-defined) source Variant.
In particular, see the following:

ST symbolic	Notes
TYPESINT	settles in a SINT ST value any numeric source Variant
TYPEINT	settles in an INT ST value any numeric source Variant
TYPEDINT	settles in a DINT ST value any numeric source Variant
TYPELINT	settles in a LINT ST value any numeric source Variant
TYPEUSINT	settles in a USINT ST value any numeric source Variant
TYPEUINT	settles in a UINT ST value any numeric source Variant
TYPEUDINT	settles in a UDINT ST value any numeric source Variant
TYPEULINT	settles in a ULINT ST value any numeric source Variant
TYPEREAL	settles in a REAL ST value any numeric source Variant
TYPELREAL	settles in a LREAL ST value any numeric source Variant
TYPETIME	settles in a TIME ST value (as numeric value) any numeric source Variant
TYPELTIME	settles in a LTIME ST value (as numeric value) any numeric source Variant
TYPECHAR	settles in a CHAR ST value (as numeric value) any numeric source Variant
TYPEWCHAR	settles in a WCHAR ST value (as numeric value) any numeric source Variant
TYPEBYTE	settles in a BYTE ST value (as numeric value) any numeric source Variant
TYPEWORD	settles in a WORD ST value (as numeric value) any numeric source Variant
TYPEDWORD	settles in a DWORD ST value (as numeric value) any numeric source Variant
TYPELWORD	settles in a LWORD ST value (as numeric value) any numeric source Variant
TYPEBOOL	settles in a BOOL ST value a **VT_BOOL** Variant
TYPESTRING	settles in a STRING ST value a **VT_BSTR** source Variant
TYPEWSTRING	settles in a WSTRING ST value a **VT_BSTR** source Variant
TYPEDATE	settles in a DATE ST value a **VT_DATE** source Variant
TYPETOD	settles in a TOD ST value a **VT_DATE** source Variant
TYPELTOD	settles in a LTOD ST value a **VT_DATE** source Variant
TYPEDT	settles in a DT ST value a **VT_DATE** source Variant
TYPELDT	settles in a LDT ST value a **VT_DATE** source Variant
TYPEVDATE	this type is explicitly forbidden to be used in assignments to ST values (from Variants); explicitly defined assignments from Variants known to be carrying DATE information, should be declared with the exact ST type they are supposed to be converted to (either **TYPEDATE**, **TYPETOD**, **TYPEDT**, **TYPELTOD** or **TYPELDT**); a **TYPEVDATE** is too generic for an ST conversion; if an automatic conversion is desired, then a **TYPEAUTO** should be used instead
TYPEVOBJECT	with this declaration the destination ST value will actually be a UDINT (unsigned 32 bits integer); it's used to explicitly state that the value is coming from a Variant source containing the reference to a COM object (a **VT_DISPATCH** carrying the pointer to the COM "Dispatch" interface); note that even though the obtained value is a UDINT, declaring it simply as a **TYPEUDINT** won't be allowed: the programmer must declare to know that the returned value is a reference to a COM object; not only to allow more strict validations over the source Variant value (when this type is declared, the source Variant will be required to be carrying a COM IDispatch), but also to mark the obtained value as one that will need special treatment: regardless their source (being it a return value from a COM method, or a value read from a COMVAR) numeric values acquired as **TYPEVOBJECT** are actually copies of references to COM objects and MUST be explicitly released with a **COMLIB_RELEASE** when no longer needed; a lax declaration as **TYPEAUTO** is allowed for this kind of values: even in these cases the programmer must be sure to know the nature of the obtained value and handle it properly

TYPECOMVAR	this declaration means that the source Variant has to be stored in a COMVAR, rather than in the destination ST value; to the destination ST value instead is assigned the value of the ID of the used COMVAR (a 32 bits unsigned integer value); this is something that makes sense in case of values returned by COM function: if the result is too complex for an ST value to be assigned directly, then it can be saved in an auxiliary COMVAR; the system always uses the same conventional auxiliary COMVAR in these cases; it's not forbidden, but makes absolutely no sense, in case of direct reads from COMVARs
TYPEARRAY	used in combination with all the types above, except TYPECOMVAR and those explicitly forbidden
TYPEAUTO	used when the script is trusted to be able to automatically decide the best match between the given COM Variant and the destination ST value (the destination type is automatically decided); the supported Variant types are the following: - **VT_I1**, **VT_I2**, **VT_INT**, **VT_I4**, **VT_I8**, **VT_UI1**, **VT_UI2**, **VT_UINT**, **VT_UI4**, **VT_DECIMAL**, **VT_UI8**, **VT_R4**, **VT_R8**: assigned to ST values of the corresponding numeric types; - **VT_BOOL**: assigned to a BOOL ST value; - **VT_BSTR**: assigned to a WSTRING ST value; - **VT_DATE**: could become either a LDT (if the Variant value is carrying both date and time information) or a TOD (if the Variant value only has a time component); - **VT_DISPATCH**: the pointer to the COM object Dispatch interface is passed on as a plain numeric value (unsigned 32 bits integer); note that if the source Variant is found to be carrying an array, the case is always treated as a **TYPECOMVAR**: the Variant is copied in a conventional auxiliary COMVAR and the ID of the COMVAR is passed on as the destination ST value (this is to avoid issues with Variant arrays potentially carrying elements with heterogeneous types); if Variant arrays are known to be compatible with ST arrays, and a straight copy in an array ST variable is desired, then their exact type should be declared, instead of using the generic **TYPEAUTO**
TYPENONE	can be used as declaration of the return type of a COM function (see **COMLIB_FXLOAD**), to explicitly state that the function will NOT return anything; must never be used while reading from COMVARs

8.2.3. Examples

The following is an example of usage of a couple of COM objects, meant to access values from an SQL database:

example

```
VAR_INPUT
   timeFROM      : STRING [100];  // IN: initial time range for query
   timeTO        : STRING [100];  // IN: final time range for query
END_VAR;
VAR_OUTPUT
   valC1_DIFF    : ULINT;         // OUT: counter 1 difference result
   valC2_DIFF    : ULINT;         // OUT: counter 2 difference result
END_VAR;
VAR
   obj_CON       : UDINT;         // COM objects
   obj_REC       : UDINT;         //
   fxCon_OPEN    : UDINT;         // COM functions
   fxCon_CLOSE   : UDINT;         //
   fxRec_OPEN    : UDINT;         //
   fxRec_GETROWS : UDINT;         //
   fxRec_CLOSE   : UDINT;         //
   strCONNECTION : STRING [256];  // DB connection string
   strQUERY      : STRING [256];  // DB query string
   varRECORDS    : UDINT;         // Actual records prepared by query
   numRECORDS    : UDINT;         // Number of records generated by the query
   numCOLUMNS    : UDINT;         // Number of columns in generated records matrix
   valC1_FROM    : ULINT;         // Local values for query processing
   valC1_TO      : ULINT;         //
   valC2_FROM    : ULINT;         //
   valC2_TO      : ULINT;         //
END_VAR;

// COM - Create the COM objects needed for DB access
obj_CON := COMLIB_LOAD ('ADODB.Connection');  // Creating an ADODB Connection object
obj_REC := COMLIB_LOAD ('ADODB.RecordSet');   // Creating an ADODB RecordSet object

// COM - Access the needed functions of the loaded COMs
// Connection.Open > has no output; needs a string (connection string) in input
fxCon_OPEN    := COMLIB_FXLOAD (obj_CON, 'Open',    typenone, typestring);
// Connection.Close > has no output; has no input
fxCon_CLOSE   := COMLIB_FXLOAD (obj_CON, 'Close',   typenone);
// RecordSet.Open > has no output; needs a string (query string) and a COM object (connection COM) in input
fxRec_OPEN    := COMLIB_FXLOAD (obj_REC, 'Open',    typenone, typestring, typevobject);
// RecordSet.Close > has no output; has no input
fxRec_CLOSE   := COMLIB_FXLOAD (obj_REC, 'Close',   typenone);
// RecordSet.GetRows > returns a variant (COMVAR) with the array of records produced by a previous query
fxRec_GETROWS := COMLIB_FXLOAD (obj_REC, 'GetRows', typecomvar);

// DB - Open database connection
strCONNECTION := 'Provider=sqloledb;Data Source=DESKTOP-OVLQ28R\SQLEXPRESS;' +
                 'Initial Catalog=MOULDING_DATA;Integrated Security=SSPI;';
COMLIB_FXCALL (fxCon_OPEN, strCONNECTION);

// Get range from input tags (maybe if not already given as function input parameters)
// timeFROM := TAG_GETVALUE ('Time_From_ UTC');  // Retrieve the initial range time from a tag
// timeTO   := TAG_GETVALUE ('Time_To_ UTC');    // Retrieve the final range time from a tag

// DB - Execute the query
//      (in our example we have a 'Date_Time' column, that we use to select a given range of records)
strQUERY := 'SELECT * FROM [MOULDING_DATA].[dbo].[ESA_Production] ' +
            'WHERE Date_Time Between $'' + timeFROM + '$' and $'' + timeTO + '$' order by Date_Time desc';
```

```
COMLIB_FXCALL (fxRec_OPEN, strQUERY, obj_CON);

// DB - Store the result of the query in a variable suitable to contain the array of records
//      (the records are placed in a COMVAR, and the ID of the COMVAR is stored in varRECORDS)
ST_OPTION HANDLE_ERRORS;  // Handle errors: the query could have generated an empty recordset
ERRORRESET();
varRECORDS := COMLIB_FXCALL (fxRec_GETROWS);  // The returned <varRECORDS> is the ID of a COMVAR
ST_OPTION BLOCKING_ERRORS;

IF ERRNO = 0 THEN

   // DB - Retrieve the size of the matrix (records x fields) generated by the query
   numRECORDS := COMVAR_GETUBOUND (varRECORDS, 2) + 1;  // Number of records = number of matrix rows
   numCOLUMNS := COMVAR_GETUBOUND (varRECORDS, 1) + 1;  // Number of fields = number of matrix columns

   // Process the values acquired from DB
   // We have a Counter1 in column 2, and a Counter2 in column 4
   // We need the difference in those counters, between the 1st and the last retrieved records
   valC1_FROM := COMVAR_GETELEMENT (varRECORDS, typeulint, numRECORDS - 1, 2);
   valC1_TO   := COMVAR_GETELEMENT (varRECORDS, typeulint, 0, 2);
   valC2_FROM := COMVAR_GETELEMENT (varRECORDS, typeulint, numRECORDS - 1, 4);
   valC2_TO   := COMVAR_GETELEMENT (varRECORDS, typeulint, 0, 4);

   valC1_DIFF := valC1_TO - valC1_FROM;
   valC2_DIFF := valC2_TO - valC2_FROM;

ELSE

   valC1_DIFF := 0;
   valC2_DIFF := 0;

END_IF;

// Save result in output tags (maybe if not provided as function output)
TAG_WRITEVALUE ('Counter1_DIFF', valC1_DIFF);
TAG_WRITEVALUE ('Counter2_DIFF', valC2_DIFF);

// DB - Close recordset and connection
COMLIB_FXCALL (fxRec_CLOSE);
COMLIB_FXCALL (fxCon_CLOSE);

// COM - Release the loaded COM functions
COMLIB_FXRELEASE (fxCon_OPEN);
COMLIB_FXRELEASE (fxCon_CLOSE);
COMLIB_FXRELEASE (fxRec_OPEN);
COMLIB_FXRELEASE (fxRec_CLOSE);
COMLIB_FXRELEASE (fxRec_GETROWS);

// COM - Release the loaded COM objects
COMLIB_RELEASE (obj_CON);
COMLIB_RELEASE (obj_REC);
```

COMLIB_LOAD

Loads and creates an instance of a COM object.

COMID = **COMLIB_LOAD** (NAME)

input		
NAME :	ANY_STRING	name of the registered COM object

output		
COMID :	UDINT	a unique numeric identifier of the created object; this ID is effectively the pointer to the IDispatch interface of the created COM object instance; this ID will have to be used in future calls to functions like **COMLIB_RELEASE** and **COMLIB_FXLOAD**

Registered COM objects, loaded from external dynamic libraries, can be created and used by the runtime only in case of Windows platforms, since COM technology won't be available in Linux systems. Note that:
- only plain functions and public properties can be referenced,
- only a subset of ST compatible types can be used with properties and functions parameters; see the 'Variant (COMVAR)' paragraphs for precise specifications about supported data types.

See the *examples* given at the beginning of this section.

COMLIB_RELEASE

Releases a previously created COM object.

COMLIB_RELEASE (COMID)

input

COMID :	UDINT	the numeric identifier of a COM object previously created by a **COMLIB_LOAD**

The function releases the COM object itself along with all the resources (allocated with it) that haven't been already explicitly cleaned up.

If still in use (still not released) all the loaded COM's functions and properties are released: this makes explicit calls to **COMLIB_FXRELEASE** completely optional (the function can still be used to make mechanics cleaner, or to handle cases where too many different functions of too many COMs have to be contemporarily used, since there is a limit to the number of COM functions available at any given time).

See the *examples* given at the beginning of this section.

COMLIB_FXLOAD

Obtains a reference to a function exported by a COM object.
Declares the data types of the values exchanged as parameters and function return.

FXID = **COMLIB_FXLOAD** (COMID, NAME [, RETURN [, PARAM1 [, PARAM2 [, ...]]]])

input

COMID :	UDINT		the numeric identifier of a COM previously loaded by a **COMLIB_LOAD**
NAME :	ANY_STRING		name of the function that has to be retrieved from the COMID object
RETURN :	UDINT	[OPTIONAL]	a type identifier code, used to declare the type of the value returned by the specified COM function; when later invoked by the **COMLIB_FXCALL**, this is the type used to build its return value; if explicitly declared as **TYPENONE** (or generally if missing), the given function is expected to have no return (expected to be <void>); it is still possible to have COM functions returning a value even if declared with no explicit return type; in these cases the return is treated as if explicitly declared as **TYPEAUTO**, and is automatically converted in the most appropriate ST type by the engine; other supported codes include: **TYPESINT**, **TYPEINT**, **TYPEDINT**, **TYPELINT**, **TYPEUSINT**, **TYPEUINT**, **TYPEUDINT**, **TYPEULINT**, **TYPEREAL**, **TYPELREAL**, **TYPETIME**, **TYPELTIME**, **TYPECHAR**, **TYPEWCHAR**, **TYPEBYTE**, **TYPEWORD**, **TYPEDWORD**, **TYPELWORD**, **TYPEBOOL**, **TYPESTRING**, **TYPEWSTRING**, **TYPEDATE**, **TYPETOD**, **TYPELTOD**, **TYPEDT**, **TYPELDT**, **TYPEVDATE**, **TYPEVOBJECT**, **TYPECOMVAR**, **TYPEARRAY**, **TYPENONE** (meaning that every type natively supported by ST can be declared, along with few common types specific of the VARIANT world); for details about automatic conversions and the best usage of these type codes, see the specifications in the 'Variant (COMVAR)' paragraphs
PARAM# :	UDINT	[OPTIONAL]	a variable number of parameters, given as type identifier codes, used to declare the types of the parameters expected by the specified COM function; if missing, it doesn't necessarily mean that the declared function needs no parameter (nor a number of parameters limited by the number of given PARAM#): any extra undeclared parameter passed to the function when later invoked for execution (**COMLIB_FXCALL**), is implicitly accepted and treated as if declared like a **TYPEAUTO**, meaning that its value is automatically handled by the ST and converted in the VARIANT type most appropriate for the given ST type; note that it is not possible to give PARAM# parameters and no RETURN parameter: if the defined function needs explicit parameters declaration and has no return, then it is mandatory to explicitly declare a **TYPENONE** return type, before the list of the needed parameters types; the usable type codes are the same listed above; again, see detailed explanations about their usage in the 'Variant (COMVAR)' paragraphs

output

FXID :	UDINT	a unique numeric identifier of the given function; this ID will have to be used in future calls to functions like **COMLIB_FXRELEASE** and **COMLIB_FXCALL**

There is a limit to the maximum number of functions that can be loaded (currently set to **128**).
If more functions are needed, then programmers might need to release some (COMLIB_FXRELEASE) before to load others.

A complete description of the supported type codes and their behaviour, along with instructions regarding how and when they should be used, is given in the 'Variant (COMVAR)' paragraphs; in particular see the '8.2.2.1. Types declaration' paragraph for general notes, the '8.2.2.2. Writing values (from ST to COM)' paragraph for specifications about passing parameters to the function, and '8.2.2.3. Reading values (from COM to ST)' paragraph for specifications about retrieving the value returned by the function.

As already stated, because of how optional parameters work, it is not possible to omit the RETURN declaration, and at the same time specify PARAM# declarations: functions with parameters and no return will have to be explicitly declared with a TYPENONE return type.
Also note that both the RETURN and the PARAM# declarations are not only optional because the function could have no return or no parameters: explicit types declarations can be omitted if the programmer trusts the ST engine to be able to automatically and properly convert the parameters and the returned value exchanged at execution time (COMLIB_FXCALL) between ST and VARIANT types. From this point of view, omitting a type declaration can be seen as leaving it virtually declared as TYPEAUTO, hence allowing it to be automatically converted by the system.
As above, precise rules of automatic conversions are given in the specifications in the 'Variant (COMVAR)' paragraphs. A precise explicit declaration of types though, for both input and output, is always strongly recommended.

When this function is used to declare COM properties, instead of COM functions, it has to be noted that:
- the type given as RETURN is the one used for conversion of values obtained from a COMLIB_PROPGET;
- the type given as PARAM1 is the one used for conversion of values passed to a COMLIB_PROPSET.

Of course, aside from extremely extravagant COM implementations, these types are generally supposed to be the same.

See the *examples* given at the beginning of this section.

COMLIB_FXRELEASE

Releases a previously loaded COM function.

COMLIB_FXRELEASE (FXID)

input

FXID :	UDINT	the numeric identifier of a function previously loaded by a **COMLIB_FXLOAD**

This method releases the function ID only.
Keep in mind that, after a release, the given ID might be reused by future **COMLIB_FXLOAD**.

See the *examples* given at the beginning of this section.

COMLIB_FXCALL

Invokes the execution of a COM object function.

[RETURN =] **COMLIB_FXCALL** (FXID [, PARAM1 [, PARAM2 [, ...]]])

input

FXID :	UDINT		the numeric identifier of a COM function previously loaded by a COMLIB_FXLOAD
PARAM# :	ANY	[OPTIONAL]	these parameters are used to give the values that have to be passed as parameters to the invoked function; their number must match that of the actual parameters of the invoked function (no parameters at all is an allowed case as well of course); the number, order and types of the parameters should have been declared with the COMLIB_FXLOAD, but the optionality of the declaration and the possibility to define TYPEAUTO parameters, allows for a lot of versatility; still, remember that, automatic or not, what matters is the result of the conversion from ST to VARIANT (see conversions rules in the 'Variant (COMVAR)' paragraphs), and what is passed with this call must be the exact list of parameters needed by the actual COM function; **failure in that might lead to malfunctions and, in the worst cases, to runtime crashes**

output

RETURN :	ANY	[OPTIONAL]	this is the value returned by the invoked COM function; formally the value could be of any type; at runtime the actual type will be the one that was declared when the function was "loaded" by the COMLIB_FXLOAD, or (in case of automatic conversion) the type that the system deems to be the most appropriate for the VARIANT received from the COM; note that if the invoked function was declared to have no return (with an explicit TYPENONE), then this COMLIB_FXCALL will have no return as well

If parameters have to be passed to the invoked function, then it is important to keep the best compatibility with the external COM needs.
See COMLIB_FXLOAD, and the detailed specifications about types and conversions in the 'Variant (COMVAR)' paragraphs, for notes about the types that can/should be used (for parameters as well as for returned values).

See the *examples* given at the beginning of this section.

COMLIB_PROPGET

Reads the value of a property member of a COM object.

RETURN = **COMLIB_PROPGET** (FXID)

input		
FXID :	UDINT	the numeric identifier of a COM property previously loaded by a **COMLIB_FXLOAD**

output		
RETURN :	ANY	this is the value currently contained in the COM property; formally the value could be of any type; at runtime the actual type will be the one that was declared as "return" type when the property was "loaded" by the **COMLIB_FXLOAD**, or (in case of implicit or explicit automatic conversion) the type that the system deems to be the most appropriate for the VARIANT received from the COM; note that the accessed property must have been declared (even implicitly) with a return type (it's impossible to read values of properties declared with a **TYPENONE** return type)

See **COMLIB_FXLOAD**, and the detailed specifications about types and conversions in the 'Variant (COMVAR)' paragraphs, for notes about the types that can/should be used.

COMLIB_PROPSET

Writes the value of a property member of a COM object.

COMLIB_PROPSET (FXID, VALUE)

input

FXID :	UDINT	the numeric identifier of a COM property previously loaded by a **COMLIB_FXLOAD**
VALUE :	ANY	this is the new value to be stored in the COM property; formally the value could be of any type; at runtime the actual type should be the one that was declared as "first parameter" type when the property was "loaded" by the **COMLIB_FXLOAD**, or (in case of implicit or explicit automatic conversion) the type appropriate for the expected automatic conversion, as specified at the beginning of this section

See **COMLIB_FXLOAD**, and the detailed specifications about types and conversions in the 'Variant (COMVAR)' paragraphs, for notes about the types that can/should be used.

COMVAR_CREATE

Creates a COMVAR (a VARIANT-like variable), usable with COM objects.

VARID = **COMVAR_CREATE** ()

output

VARID :	UI32	the ID of the created COMVAR

As explained at the beginning of this section, we call a "COMVAR" a special object used by ST to encapsulate the functionalities of the VARIANT data used by COM objects and interfaces.
This function creates one of these objects and returns its unique identifier. Every further access and manipulation of that object will require the programmer to specify this ID in order to identify the specific COMVAR that needs to be used.

Remember that after a COMVAR object has been explicitly created by this function, it will have to be cleaned up and released after use (see COMVAR_DESTROY: there is no limit to the number of usable COMVARs, but the related system resources must be freed when no longer needed).

Explicit vs Implicit COMVARs

Not only COMVAR objects can be explicitly created by this function: they could also be implicitly generated by invocations of COM functions (COMLIB_FXCALL) declared to return a TYPECOMVAR, or when values are acquired from similarly declared COM properties (COMLIB_PROPGET).
These implicit generations will not cause the creation of multiple COMVARs: instead one only system COMVAR is dedicated to their storage, and new generations will simply overwrite old ones.
This means a couple of things the programmer is supposed to take note of:
- all these COMVARs will have the same ID (specifically their ID is always supposed to be 0, but programmers should not rely on this constant);
- these COMVARs are NOT supposed to be cleaned up: overwritten values are automatically released by the system when needed; trying to release these COMVARs will result in errors;
- since new generated COMVARs overwrite old ones, if objects returned by multiple calls of COM functions and properties have to be retained, the programmer might need to explicitly copy (COMVAR_COPY) them in other purposely created objects (an example is given below).

example

```
VAR
   com_obj : UDINT;
   com_fx  : UDINT;
   com_var : UDINT;
   idx     : UDINT;
   com_var_n : ARRAY [10] OF UDINT;
END_VAR;

com_obj := COMLIB_LOAD   ('ComOBJ');                      // A COM object
com_fx  := COMLIB_FXLOAD (com_obj, 'ComFX', typecomvar);  // A function returning a VARIANT (COMVAR)

// 1) If there is no need to keep the values

com_var := COMLIB_FXCALL (com_fx);                        // Multiple calls will always store the result
com_var := COMLIB_FXCALL (com_fx);                        // in the same COMVAR and will always return
com_var := COMLIB_FXCALL (com_fx);                        // the same COMVAR ID (0)
com_var := COMLIB_FXCALL (com_fx);                        // There is no need to release these COMVARs
```

```
// 2) If there is the need to keep and use multiple returned values

// - gather the values
FOR idx := 0 TO 9 DO
   com_var := COMLIB_FXCALL (com_fx);         // call the COM function expected to return the VARIANT
   com_var_n[idx] := COMVAR_CREATE ();        // create a COMVAR dedicated to the received value
   COMVAR_COPY (com_var, com_var_n[idx]);     // store the received value as a copy
END_FOR;

   // - do whatever is needed with the saved values …

// - release the values
FOR idx := 0 TO 9 DO
   COMVAR_DESTROY (com_var_n[idx]);           // destroy every copy explicitly created
END_FOR;

COMLIB_FXRELEASE (com_fx);
COMLIB_RELEASE (com_obj);
```

COMVAR_DESTROY

Cleans up and releases an existing COMVAR.

COMVAR_DESTROY (VARID)

input

VARID :	UDINT	the numeric identifier of a COMVAR object previously created by a **COMVAR_CREATE**; never to be used with IDs of COMVARs implicitly generated by COM invocations

Programmers must call a matching **COMVAR_DESTROY** for every **COMVAR_CREATE** explicitly called.
After destruction, the given ID might be reused and returned by future **COMVAR_CREATE**.

COMVARs implicitly generated as return values by COM invocations through **COMLIB_FXCALL** and **COMLIB_PROPGET** don't need destruction; they are automatically taken care of by the system every time new invocations and new generations occur.
See details about "implicit" COMVARs among the notes given above for the **COMVAR_CREATE** function. See the given example as well.

COMVAR_CLEANUP

Cleans up the allocated value of an existing COMVAR.

COMVAR_CLEANUP (VARID)

input

VARID :	UDINT	the numeric identifier of a COMVAR object previously created by a **COMVAR_CREATE**, or implicitly generated by COM invocations

Similar to a **COMVAR_DESTROY** from the point of view of the release of the value resources, but this function only affects the value of the COMVAR, not the existence of the object itself.
The memory used is released and the value of the COMVAR is lost, but the COMVAR object itself still exists after the cleanup; its VARID is retained and further **COMVAR_CREATE** won't be needed in order to use it again.

Note that this operation is not explicitly needed before a **COMVAR_DESTROY**, a **COMVAR_COPY**, or a **COMVAR_DIMARRAY**, since all these kinds of access automatically cleanup the old COMVAR value.

COMVAR_COPY

Copies the value from an existing COMVAR to another.

COMVAR_COPY (SOURCEID, DESTID)

input		
SOURCEID :	UDINT	the numeric identifier of the source COMVAR object
DESTID :	UDINT	the numeric identifier of the destination COMVAR object

Both the referenced COMVAR objects must exist; both could have been created by an explicit **COMVAR_CREATE**, or implicitly generated by COM invocations.
This function automatically cleans up the content of the destination COMVAR and stores in it a copy of the source one. A preemptive explicit **COMVAR_CLEANUP** of the destination is not needed.

COMVAR_DIMARRAY

Initializes an existing COMVAR value, making it a VARIANT-compatible array with given dimensions.

COMVAR_DIMARRAY (VARID, LOW1, HIGH1 [, LOW2, HIGH2 [, ...]])

input			
VARID :	UDINT		the numeric identifier of a COMVAR object previously created by a **COMVAR_CREATE**, or implicitly generated by COM invocations
LOW# :	UDINT	[OPTIONAL]	the lower bound of an Nth dimension of the array; must always be ≤ HIGH# formally optional, but at least an initial LOW1 must exist; note that LOW# and HIGH# are always given in pairs
HIGH# :	UDINT	[OPTIONAL]	the upper bound of an Nth dimension of the array; must always be ≥ LOW# formally optional, but at least an initial HIGH1 must exist; note that LOW# and HIGH# are always given in pairs

This function is used to prepare a COMVAR to make it contain an array able to store elements compatible with any kind of VARIANT value.
The function automatically cleans up the old content of the referenced COMVAR and stores in it simply an empty array, that could be filled up later by accessing its individual elements. A preemptive explicit **COMVAR_CLEANUP** of the COMVAR is not needed.

The created array could be defined with any number of dimensions.
A pair of bounds (LOW#, HIGH#) should be specified for each desired dimension.
Bounds for the 1st dimension are mandatory (since the array must have at least 1 dimension); further pairs are optional, and depend on the actual need for the extra dimensions.

COMVAR_GETNUMDIM

Retrieves the number of dimensions of a COMVAR array.

NUMDIM = **COMVAR_GETNUMDIM** (VARID)

input		
VARID :	UDINT	the numeric identifier of an existing COMVAR object; the COMVAR is supposed to have a value of ARRAY type

output		
NUMDIM :	UDINT	the number of dimensions of the array contained in the referenced COMVAR

Using this function on a non-array COMVAR results in failure.

Using this function in combination with **COMVAR_GETLBOUND** and **COMVAR_GETUBOUND**, allows to browse the definition of all the dimensions and sizes of any kind of array.

COMVAR_GETLBOUND

Retrieves the lower bound of a given dimension of a COMVAR array.

LBOUND = **COMVAR_GETLBOUND** (VARID, DIM)

input		
VARID :	UDINT	the numeric identifier of an existing COMVAR object; the COMVAR is supposed to have a value of ARRAY type
DIM :	UDINT	the (base-1) index of a dimension of the array; the index must be: 1 ≤ DIM ≤ N (being N the number of dimensions of the referenced array)
output		
LBOUND :	UDINT	the lower bound of the given dimension of the given COMVAR array

The referenced COMVAR must contain a valid array.
The specified DIM must match the range of the dimensions of the array.

COMVAR_GETUBOUND

Retrieves the upper bound of a given dimension of a COMVAR array.

UBOUND = **COMVAR_GETUBOUND** (VARID, DIM)

input		
VARID :	UDINT	the numeric identifier of an existing COMVAR object; the COMVAR is supposed to have a value of ARRAY type
DIM :	UDINT	the (base-1) index of a dimension of the array; the index must be: 1 ≤ DIM ≤ N (being N the number of dimensions of the referenced array)
output		
UBOUND :	UDINT	the upper bound of the given dimension of the given COMVAR array

The referenced COMVAR must contain a valid array.
The specified DIM must match the range of the dimensions of the array.

COMVAR_SET

Writes a value in a COMVAR.

COMVAR_SET (VARID, TYPE, VALUE)

input

VARID :	UDINT	the numeric identifier of an existing COMVAR object
TYPE :	UDINT	a type identifier code, used to declare the type of the value assigned to the COMVAR (formally the type intended for the destination VARIANT created as result of the assignment); note that this parameter is never optional; even in case of automatic ST-VARIANT conversions, at least an explicit **TYPEAUTO** should be given; see detailed specifications regarding type codes, their usage and behaviour, in the 'Variant (COMVAR)' paragraphs; in particular, see notes from the '8.2.2.2. Writing values (from ST to COM)' paragraph, where directions are given for conversions applied when ST values are transformed in COM VARIANT values; see also the comments given for the **COMLIB_FXLOAD** function, where the declaration of the function parameters is described
VALUE :	ANY	the new value to be assigned to the COMVAR; formally defined as "ANY", the type of this parameter at runtime should match the definition given as TYPE above

The function automatically cleans up the old content of the referenced COMVAR, before the storage of the new value takes effect. A preemptive explicit **COMVAR_CLEANUP** of the COMVAR is not needed.

COMVAR_SETELEMENT

Writes a value in an element of a COMVAR array.

COMVAR_SETELEMENT (VARID, TYPE, VALUE, INDEX1 [, INDEX2 [, ...]])

input

VARID :	UDINT		the numeric identifier of an existing COMVAR object; the COMVAR is supposed to already have a value of ARRAY type
TYPE :	UDINT		a type identifier code, used to declare the type of the value assigned to the COMVAR element (formally the type intended for the destination VARIANT element created as result of the assignment); note that this parameter is never optional; even in case of automatic ST-VARIANT conversions, at least an explicit **TYPEAUTO** should be given; see detailed specifications regarding type codes, their usage and behaviour, in the 'Variant (COMVAR)' paragraphs; in particular, see notes from the '8.2.2.2. Writing values (from ST to COM)' paragraph, where directions are given for conversions applied when ST values are transformed in COM VARIANT values; see also the comments given for the **COMLIB_FXLOAD** function, where the declaration of the function parameters is described
VALUE :	ANY		the new value to be assigned to the COMVAR element; formally defined as "ANY", the type of this parameter at runtime should match the definition given as TYPE above
INDEX# :	UDINT	[OPTIONAL]	a variable number of parameters used to specify the indexes (base-0) of the targeted element; there should be one INDEX# parameter for each dimension of the referenced COMVAR array

The referenced COMVAR must already contain a valid array value.
This function merely replaces the value of one of its elements.
See **COMVAR_DIMARRAY** for hints about how to prepare a COMVAR with an initialized empty array.

COMVAR_GET

Reads the value from a COMVAR.

VALUE = **COMVAR_GET** (VARID, TYPE)

input		
VARID :	UDINT	the numeric identifier of an existing COMVAR object
TYPE :	UDINT	a type identifier code, used to declare the type of the value read from the COMVAR (formally the type intended for the destination ST value created as result of the assignment); note that this parameter is never optional; even in case of automatic VARIANT-ST conversions, at least an explicit **TYPEAUTO** should be given; see detailed specifications regarding type codes, their usage and behaviour, in the 'Variant (COMVAR)' paragraphs; in particular, see notes from the '8.2.2.3. Reading values (from COM to ST)' paragraph, where directions are given for conversions applied when COM VARIANT values are transformed in ST values; see also the comments given for the **COMLIB_FXLOAD** function, where the declaration of the function return is described
output		
VALUE :	ANY	the value read from the COMVAR to be assigned to the destination ST; formally defined as "ANY", the type of this parameter at runtime should match the definition given as TYPE above

COMVAR_GETELEMENT

Reads the value from an element of a COMVAR array.

VALUE = **COMVAR_GETELEMENT** (VARID, TYPE, INDEX1 [, INDEX2 [, ...]])

input			
VARID :	UDINT		the numeric identifier of an existing COMVAR object the COMVAR is supposed to contain a value of ARRAY type
TYPE :	UDINT		a type identifier code, used to declare the type of the value read from the COMVAR (formally the type intended for the destination ST value created as result of the assignment); note that this parameter is never optional; even in case of automatic VARIANT-ST conversions, at least an explicit **TYPEAUTO** should be given; see detailed specifications regarding type codes, their usage and behaviour, in the 'Variant (COMVAR)' paragraphs; in particular, see notes from the '8.2.2.3. Reading values (from COM to ST)' paragraph, where directions are given for conversions applied when COM VARIANT values are transformed in ST values; see also the comments given for the **COMLIB_FXLOAD** function, where the declaration of the function return is described
INDEX# :	UDINT	[OPTIONAL]	a variable number of parameters used to specify the indexes (base-0) of the targeted element; there should be one INDEX# parameter for each dimension of the referenced COMVAR array
output			
VALUE :	ANY		the value read from the COMVAR element, to be assigned to the destination ST; formally defined as "ANY", the type of this parameter at runtime should match the definition given as TYPE above

< VARIABLES >

The following are the variables usable to share information and directives for the libraries management:

COMLIB errors

Some COM objects are able to provide detailed error information about the execution of their own methods; with the following couple of variables it's possible to retrieve the meaningful parts.

COMLIB_ERRNO

type UDINT
access R/W

gives the result code of the last invoked COM method or property (filled up after a COMLIB_FXCALL, COMLIB_PROPGET or COMLIB_PROPSET call); coding conventions follow the Microsoft COM standards;
always matches the error message stored in COMLIB_ERRMSG (see below);
0 (zero) indicates a successful invocation; anything else is an error;
note that in case of error, the standard ST error management and variables will be operating too, so meaningful (but less detailed) information could be retrieved from the standard FXRESULT, ERRNO and ERRMSG variables; also note that this kind of information is accessible only if the option ST_OPTION HANDLE_ERRORS is currently set, otherwise the error presence would halt the script execution;
important: this variable is set only if error information are actually provided by the COM object and should be intended as an extension of the standard error information; because of this, and since errors could happen at different levels as well, programmers should not rely on this value to know whether errors happened or not;

COMLIB_ERRMSG

type WSTRING
access R/W

gives a textual description of the error currently stored in COMLIB_ERRNO (see above for details);
in case of successful executions this variable is reset to an empty string

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
COMLIBNOFX	0xFFFFFFFF	*** COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$ (returned by COMLIB_FXLOAD along with an error in case nothing was loaded)
TYPESINT	1	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEINT	2	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEDINT	3	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELINT	4	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEUSINT	5	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEUINT	6	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEUDINT	7	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEULINT	8	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEREAL	9	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELREAL	10	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPETIME	11	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELTIME	12	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEDATE	13	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPETOD	14	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELTOD	15	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEDT	16	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELDT	17	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPESTRING	18	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEWSTRING	19	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPECHAR	20	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEWCHAR	21	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEBOOL	22	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEBYTE	23	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEWORD	24	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEDWORD	25	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPELWORD	26	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEVDATE	0x1007	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEVOBJECT	0x1009	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPECOMVAR	0x100C	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEARRAY	0x2000	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPEAUTO	0x0FFE	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$
TYPENONE	0x0FFF	COMLIB_FXLOAD, COMLIB_FXCALL, COMVAR_SET$$$, COMVAR_GET$$$

9. COMMON - PRINT

Printing

<TODO>

PDF

Few general notes about PDF documents production:
- only one document at a time can be produced; old documents must be closed (**PDF_CLOSE**) before new ones can be created (**PDF_OPEN**);
- most of the given measures (coordinates and sizes) are given in units equal to 1/72nd of inch (given in floating-point, so fractional measures are allowed);
- the only exception is the size of the fonts, which is expressed in points when selected (**PDF_SETFONTSIZE**; again fractional measures are allowed); note that there is no automatic relation between the fonts sizes in points, and the corresponding physical measures in 72nd of inch, since it depends entirely on the definition of the fonts themselves (for example the height of a "Tunga" string of a given number of points is almost twice the height of a "Stencil" string of the same number of points);
- the coordinates (0,0) correspond to the upper-left corner of the document page; positive axis go towards the right of the page and towards the bottom of the page, so that the coordinates of the bottom-right corner correspond to the size of the page itself (**PDF_PAGEWIDTH**,**PDF_PAGEHEIGHT**);
- selected colors affect all the drawing directives (both geometric and texts);
- selected line sizes affect all geometric drawing directives (make no sense for texts).

PDF_OPEN

Starts the creation of a new PDF document.

PDF_OPEN (FILENAME [, PWDUSER [, PROTCTX [, PWDOWNER]]])

input

FILENAME :	ANY_STRING		name of the newly created PDF file
PWDUSER :	ANY_STRING	[OPTIONAL]	protection parameter: user password; used to apply encryption and a password to the document; users will have to enter it to open the document
PROTCTX :	ANY_INT	[OPTIONAL]	protection parameter: protected contexts flags; used to specify a list of management contexts that can be disabled in the produced document; see the flags list below
PWDOWNER :	ANY_STRING	[OPTIONAL]	protection parameter: owner password used to specify a further password that users can give to re-enable the contexts disabled by the parameter above

Only one PDF document at a time can be produced.
This function can be used only if no other document is currently under construction.
Creations started with this method are expected to be closed and finalized with a call to **PDF_CLOSE**.

The created document starts with:
- an empty page, in vertical layout, size A4,

embedding the following defaults:
- a BLACK color for all drawing directives,
- a line width (for geometric directives) of 1/72nd of inch,
- an ARIAL (normal) font, with a size of 10 points (no bold/italic/underline attributes).

As soon as the document is created, the variables **PDF_PAGEWIDTH** and **PDF_PAGEHEIGHT** assume the values of the default page size (A4), while the variables **PDF_FONTHEIGHT**, **PDF_FONTASCENT** and **PDF_FONTDESCENT** assume the values of the corresponding metrics of the default font (Arial 10pt).

Protection

If the optional protection parameters are given, then the document is produced encrypted, and different contexts can be chosen to be disabled or protected by password:
- the PWDUSER is a password that will be needed to open the document;
- the PROTCTX is a bitstring of flags that can be used to specify different contexts of the document management (such as the ability to print, edit, add comments, and so on) that have to be disabled;
- the PWDOWNER is a password that can be used to re-enable the contexts that could have been disabled by the flags above;
  this owner password will have to be entered by the user when the document is opened, and that will only be possible if also a valid (not empty) PWDUSER was provided for the document creation.

So, for the different possible combination of the provided protection parameters:
- (FILENAME)
  if no optional parameter is given, then no encryption and no restriction will be applied;
- (FILENAME, PWDUSER)
  if only the PWDUSER is given, the document will require a password to be opened;
  no further restriction will be applied;
  if an empty password is provided, then no protection is applied, as if the parameter were missing;
- (FILENAME, PWDUSER, PROTCTX)
  if also PROTCTX flags are given, then the specified contexts will be disabled;
  the PWDUSER parameter will follow its own independent rules, as above, and can be given either as a valid or as an empty password;
- (FILENAME, PWDUSER, PROTCTX, PWDOWNER)
  if also the PWDOWNER is given, then the user will have the chance to unlock the contexts protected by the PROTCTX flags;
  this effect will be strictly dependent on the existence of the first two parameters: the 'unlock' action will only have effect on the contexts explicitly disabled by the PROTCTX, and will only be possible if a password will be asked to the user, which - again - will only happen if a valid PWDUSER was given as well;
  in other words, all three parameters are required: when the document is opened, a password is asked; if the PWDUSER is entered, then the opened document will have its selected contexts disabled; if the PWDOWNER is entered instead, those contexts will be enabled.

As for the list of the acceptable flags that can be specified in the PROTCTX parameter (combined in 'OR' in a bitstring), according to the PDF standards, the following are the supported values (see in the <CONSTANTS> section the corresponding numeric values if needed):

PDFCTXPRINT	print the document
PDFCTXEDIT	modify the document besides annotations, form fields or changing pages
PDFCTXCOPY	extract text and graphic
PDFCTXEDITNOTES	add or modify text annotations or form fields
PDFCTXFILLANDSIGN	fill in existing form or signature fields
PDFCTXACCESSIBLE	extract text and graphics to support user with disabilities
PDFCTXDOCASSEMBLY	assemble the document: insert, create, rotate delete pages or add bookmarks
PDFCTXHIGHPRINT	print a high-resolution version of the document
PDFCTXALL	combine all the contexts above; only plain read will be allowed

PDF_CLOSE

Finalizes the production of the current PDF document.

PDF_CLOSE ()

Only usable when a PDF document is actually being produced, started with a previous call to **PDF_OPEN**.
After this function has been used to close a document, a further open can be invoked to create another.

As soon as the document is closed, the variables **PDF_PAGEWIDTH**, **PDF_PAGEHEIGHT**, **PDF_FONTHEIGHT**, **PDF_FONTASCENT** and **PDF_FONTDESCENT** are automatically reset to 0.

PDF_NEWPAGE

Adds a new page to the PDF document.

PDF_NEWPAGE ()

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The old page is left and won't be accessible anymore.
All subsequent drawing directives will be part of the new page.

PDF_SETCOLOR

Sets a new color, to be used in all subsequent drawing directives.

PDF_SETCOLOR (COLOR)

input

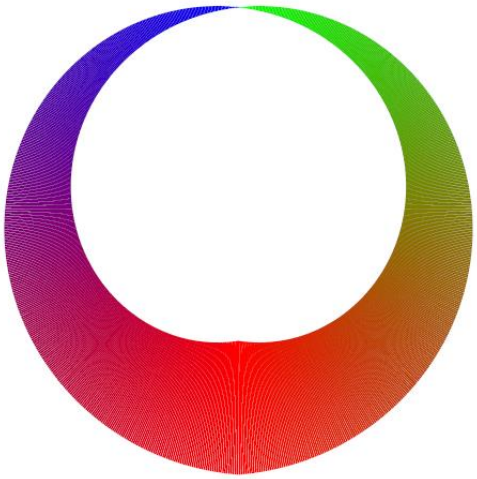
COLOR :	UDINT	the new color, given in RGB form: 0x00BBGGRR

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given color will be valid for both geometric and text directives.

See the following *example* for a combined usage of colors and lines
(the given example draws a series of lines, in a circular pattern, of variable size and colors):

example

```
VAR
  rad   : LREAL := 60.0;
  steps : LREAL := 400.0;
  astep : LREAL;
  wstep : LREAL;
  ang   : LREAL;
  wdt   : LREAL;
  cs1, sn1, cs2, sn2 : LREAL;
  rgb1, rgb2, cmp : UDINT;
END_VAR;

PDF_OPEN ('D:\stpdf.pdf');

astep := _PI / steps;
ang := 0;
wstep := 40.0 / steps;
wdt := 0;
WHILE (ang <= _PI) DO
  cs2 := COS(ang) * rad;
  sn2 := SIN(ang) * rad;
  IF (ang > 0) THEN
    PDF_SETLINEWIDTH (wdt);
    cmp := ANY_TO_UDINT (ang / _PI * 255);
    rgb1 := cmp + (255-cmp) * 256;
    rgb2 := cmp + (255-cmp) * 65536;
    PDF_SETCOLOR (rgb1);
    PDF_DRAWLINE (100 + sn1, 150 - cs1, 100 + sn2, 150 - cs2);
    PDF_SETCOLOR (rgb2);
    PDF_DRAWLINE (100 - sn1, 150 - cs1, 100 - sn2, 150 - cs2);
  END_IF;
  cs1 := cs2;
  sn1 := sn2;
  ang := ang + astep;
  wdt := wdt + wstep;
END_WHILE;

PDF_CLOSE ();
```



PDF_SETLINEWIDTH

Sets a new line width, to be used in all subsequent geometric drawing directives.

PDF_SETLINEWIDTH (WIDTH)

input		
WIDTH :	ANY_NUM	the new width, given in 72nd of inch; allowed to specify even fractional parts of the measure unit

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).

See a usage *example* given along with the **PDF_SETCOLOR** description.

PDF_SETFONT

Sets up a whole new font, with a full set of given attributes.

PDF_SETFONT (FONT, SIZE, BOLD, ITALIC, UNDERLINE)

input

FONT :	ANY_STRING	the name of the font
SIZE :	ANY_NUM	the size of the font (in points)
BOLD :	BOOL	the font bold attribute
ITALIC :	BOOL	the font italic attribute
UNDERLINE :	BOOL	the font underline attribute

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new font.
Usable when several properties of the font need their selection; if only single attributes need an adjustment, dedicated functions are available as well (see **PDF_SETFONT$$$** below).

As soon as the selection is done, the variables **PDF_FONTHEIGHT**, **PDF_FONTASCENT** and **PDF_FONTDESCENT** assume the values of the corresponding metrics of the selected font.

Note that font selections might weight a lot in the final size of the produced PDF file.

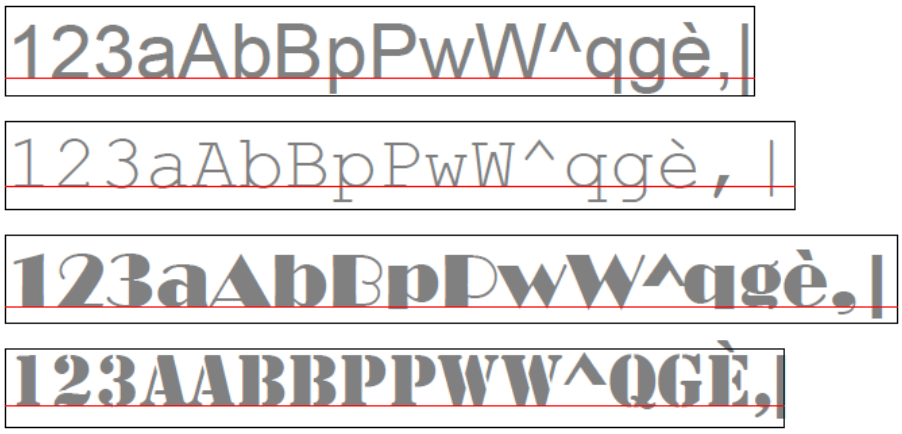
See the following *example* for a combined usage of fonts, metrics and geometric directives:

example

```
VAR
  xx : LREAL := 100.0;
  yy : LREAL := 100.0;
  st : WSTRING [20] := "123aAbBpPwW^qgè,|";
END_VAR;

FUNCTION PdfOutText                                    // gathers a batch of instruction
                                                       // for a text output
  VAR_INPUT
    size  : LREAL;
  END_VAR;
  VAR
    width : LREAL;
  END_VAR;

  PDF_SETFONTSIZE (size);                              // set the size of the font
  PDF_SETCOLOR (16#808080);                            // draw texts in gray
  PDF_DRAWTEXT (xx, yy, st);
  width := PDF_GETTEXTWIDTH (st);                      // get text width for frame
  PDF_SETCOLOR (16#000000);                            // draw rectangle in black
  PDF_DRAWRECTANGLE (xx, yy, xx + width, yy + PDF_FONTHEIGHT, FALSE);
  PDF_SETCOLOR (16#0000FF);                            // draw base line in red
  PDF_DRAWLINEH (xx, yy + PDF_FONTASCENT, width);
  yy := yy + PDF_FONTHEIGHT + 10;

END_FUNCTION;

PDF_OPEN ('D:\stpdf.pdf');
PDF_SETLINEWIDTH (0.1);                                // thin frame and base line

PdfOutText (30);                                       // text 1: arial (default)
```

```
PDF_SETFONTNAME ("Courier New");
PdfOutText (30);                                    // text 2: courier
PDF_SETFONTNAME ("Broadway");
PdfOutText (30);                                    // text 3: broadway
PDF_SETFONTNAME ("Stencil");
PdfOutText (30);                                    // text 4: stencil

PDF_CLOSE ();
```

the given example draws 4 texts with different fonts (the 1st being the default Arial), with same size (30), surrounds them with a rectangular frame, marking their exact dimensions, and outlines their base lines, as given by the known font metrics:



PDF_SETFONTNAME

Selects a new font.

PDF_SETFONTNAME (FONT)

input		
FONT :	ANY_STRING	the name of the font

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new font.
Only the font facename is given to select a whole new font; the other font attributes are preserved from the last dedicated selections.

As soon as the selection is done, the variables **PDF_FONTHEIGHT**, **PDF_FONTASCENT** and **PDF_FONTDESCENT** assume the values of the corresponding metrics of the selected font.

Note that font selections might weight a lot in the final size of the produced PDF file.

See a usage *example* given along with the **PDF_SETFONT** description.

PDF_SETFONTSIZE

Selects a new font size.

PDF_SETFONTSIZE (SIZE)

input		
SIZE :	ANY_NUM	the size of the font (in points)

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new size.
Only the font size is affected; the other font attributes are preserved from the last dedicated selections.

As soon as the selection is done, the variables **PDF_FONTHEIGHT**, **PDF_FONTASCENT** and **PDF_FONTDESCENT** assume the values of the corresponding metrics of the selected font.

Note that font selections might weight a lot in the final size of the produced PDF file.

See a usage *example* given along with the **PDF_SETFONT** description.

PDF_SETFONTBOLD

Selects a new font bold attribute.

PDF_SETFONTBOLD (BOLD)

input		
BOLD :	BOOL	the font bold attribute

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new attribute state.
Only the given state is affected; the other font attributes are preserved from the last dedicated selections.

Note that font selections might weight a lot in the final size of the produced PDF file.

PDF_SETFONTITALIC

Selects a new font italic attribute.

PDF_SETFONTITALIC (ITALIC)

input		
ITALIC :	BOOL	the font italic attribute

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new attribute state.
Only the given state is affected; the other font attributes are preserved from the last dedicated selections.

Note that font selections might weight a lot in the final size of the produced PDF file.

PDF_SETFONTUNDERLINE

Selects a new font underline attribute.

PDF_SETFONTUNDERLINE (UNDERLINE)

input		
UNDERLINE :	BOOL	the font underline attribute

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
All subsequent calls to text drawings will use the new attribute state.
Only the given state is affected; the other font attributes are preserved from the last dedicated selections.

Note that font selections might weight a lot in the final size of the produced PDF file.

PDF_DRAWTEXT

Draws a text in the PDF document.

PDF_DRAWTEXT (POSX, POSY, TEXT)

input		
POSX :	ANY_NUM	X coordinate of the left margin of the drawn text; given in 72nd of inch
POSY :	ANY_NUM	Y coordinate of the top margin of the drawn text; given in 72nd of inch
TEXT :	ANY_STRING	the text string drawn in the document

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given text will be drawn in the current page of the document, at the given coordinates (upper left corner), using the current color and the current font.

See a usage *example* given along with the **PDF_SETFONT** description.

PDF_DRAWLINE

Draws a line in the PDF document.

PDF_DRAWLINE (FROMX, FROMY, TOX, TOY)

input		
FROMX :	ANY_NUM	X coordinate of the initial segment point; given in 72nd of inch
FROMY :	ANY_NUM	Y coordinate of the initial segment point; given in 72nd of inch
TOX :	ANY_NUM	X coordinate of the final segment point; given in 72nd of inch
TOY :	ANY_NUM	Y coordinate of the final segment point; given in 72nd of inch

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given line will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width.

See a usage *example* given along with the **PDF_SETCOLOR** description.

PDF_DRAWLINEH

Draws a horizontal line in the PDF document.

PDF_DRAWLINEH (FROMX, FROMY, LENGTH)

input

FROMX :	ANY_NUM	X coordinate of the initial segment point; given in 72nd of inch
FROMY :	ANY_NUM	Y coordinate of the initial segment point; given in 72nd of inch
LENGTH :	ANY_NUM	total length of the drawn segment; given in 72nd of inch; can be negative to draw lines towards lower coordinates (towards the right of the page)

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given line will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width.

PDF_DRAWLINEV

Draws a vertical line in the PDF document.

PDF_DRAWLINEV (FROMX, FROMY, LENGTH)

input		
FROMX :	ANY_NUM	X coordinate of the initial segment point; given in 72nd of inch
FROMY :	ANY_NUM	Y coordinate of the initial segment point; given in 72nd of inch
LENGTH :	ANY_NUM	total length of the drawn segment; given in 72nd of inch; can be negative to draw lines towards lower coordinates (towards the top of the page)

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given line will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width.

PDF_DRAWRECTANGLE

Draws a rectangle in the PDF document.

PDF_DRAWRECTANGLE (FROMX, FROMY, TOX, TOY, FILL)

input

FROMX :	ANY_NUM	X coordinate of a corner of the rectangle; given in 72nd of inch
FROMY :	ANY_NUM	Y coordinate of the same (FROMX) corner of the rectangle; given in 72nd of inch
TOX :	ANY_NUM	X coordinate of the opposite (to FROM) corner of the rectangle; given in 72nd of inch
TOY :	ANY_NUM	Y coordinate of the opposite (to FROM) corner of the rectangle; given in 72nd of inch
FILL :	BOOL	FALSE if only the empty border of the rectangle has to be drawn; TRUE if a filled rectangular box has to be drawn

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given rectangle will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width (line width being meaningful in case of empty shape only).

PDF_DRAWCIRCLE

Draws a circle in the PDF document.

PDF_DRAWCIRCLE (CENTERX, CENTERY, RADIUS, FILL)

input

CENTERX :	ANY_NUM	X coordinate of the center of the circle; given in 72nd of inch
CENTERY :	ANY_NUM	Y coordinate of the center of the circle; given in 72nd of inch
RADIUS :	ANY_NUM	radius of the circle; given in 72nd of inch
FILL :	BOOL	FALSE if only an empty circumference has to be drawn; TRUE if a filled circular disc has to be drawn

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given circle will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width (line width being meaningful in case of empty shape only).

PDF_DRAWELLIPSE

Draws an ellipse in the PDF document.

PDF_DRAWELLIPSE (FROMX, FROMY, TOX, TOY, FILL)

input

FROMX :	ANY_NUM	X coordinate of a corner of the rectangle that contains the ellipse (say the X coordinate of the leftmost point of the ellipse); given in 72nd of inch
FROMY :	ANY_NUM	Y coordinate of the same (FROMX) corner of the containing rectangle (say the Y coordinate of the uppermost point of the ellipse); given in 72nd of inch
TOX :	ANY_NUM	X coordinate of the opposite (to FROM) corner of the rectangle (say the X coordinate of the rightmost point of the ellipse); given in 72nd of inch
TOY :	ANY_NUM	Y coordinate of the opposite (to FROM) corner of the rectangle (say the Y coordinate of the lowermost point of the ellipse); given in 72nd of inch
FILL :	BOOL	FALSE if only the empty border of the ellipse has to be drawn; TRUE if a filled elliptical disc has to be drawn;

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).
The given ellipse will be drawn in the current page of the document, at the given coordinates, using the current color and the current line width (line width being meaningful in case of empty shape only).

PDF_DRAWIMAGE

Draws an image in the PDF document.

PDF_DRAWIMAGE (POSX, POSY, WIDTH, HEIGHT, FILENAME)

input

POSX :	ANY_NUM	X coordinate of the upper-left corner of the image; given in 72nd of inch
POSY :	ANY_NUM	Y coordinate of the upper-left corner of the image; given in 72nd of inch
WIDTH :	ANY_NUM	width of the drawn image; given in 72nd of inch; can be 0 (together with HEIGHT) for an automatic management of the size
HEIGHT :	ANY_NUM	height of the drawn image; given in 72nd of inch; can be 0 (together with WIDTH) for an automatic management of the size
FILENAME :	ANY_STRING	name of the file containing the needed image

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).

If a size of (0,0) is given, then the size is handled automatically:
the system keeps the original size of the image (as given by the imported file) as long as there is enough room in the page, starting from the given position;
if there is not enough room, then the image is scaled down to fit within the page borders, keeping the original aspect ratio.

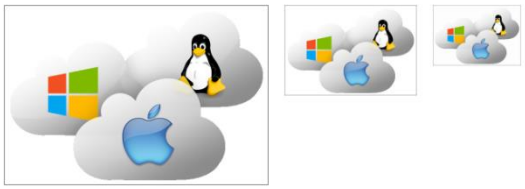
The images implementation only supports PNG and JPG formats.

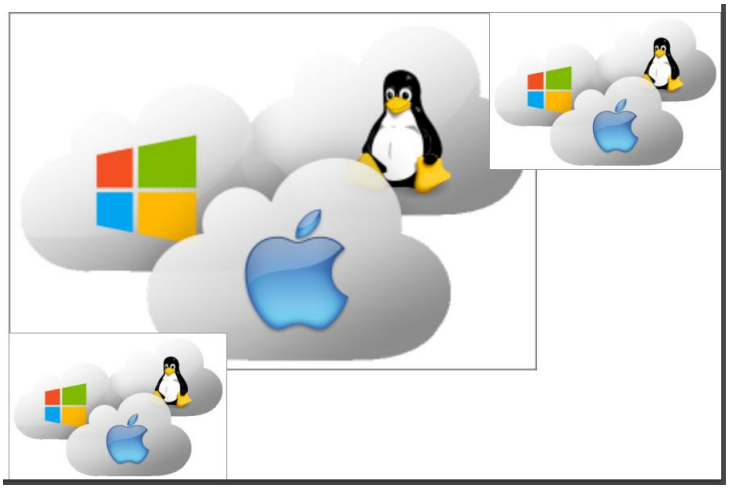
See the following *example* as an illustration of how to manage pictures, sizes and automatic stretches:

example

```
VAR
   pw : LREAL := 330;
   ph : LREAL := 223;
   px : LREAL := 100.0;
   py : LREAL := 100.0;
END_VAR;

PDF_OPEN ('D:\stpdf.pdf');

pw := pw / 2;
ph := ph / 2;
PDF_DRAWIMAGE (px, py, pw, ph, 'D:\pic.png');       // Draw few images with different sizes
px := px + pw + 10;
PDF_DRAWIMAGE (px, py, pw/2, ph/2, 'D:\pic.png');
px := px + pw/2 + 10;
PDF_DRAWIMAGE (px, py, pw/3, ph/3, 'D:\pic.png');
px := px + pw/3 + 10;

PDF_DRAWIMAGE (150, 550, 0, 0, 'D:\pic.png');       // draw images with automatic size
PDF_DRAWIMAGE (450, 550, 0, 0, 'D:\pic.png');
PDF_DRAWIMAGE (150, 750, 0, 0, 'D:\pic.png');

PDF_CLOSE ();
```

the given example shows how images can be scaled when drawn in the document, and how they can be automatically resized when near page edges





PDF_GETTEXTWIDTH

Retrieves the width of a given text.

WIDTH = **PDF_GETTEXTWIDTH** (TEXT)

input		
TEXT :	ANY_STRING	text to be measured

output		
WIDTH :	LREAL	calculated text width; given in 72nd of inch

Only usable if a PDF document is actually being produced (see **PDF_OPEN**).

The size calculated and returned corresponds to the width of the given text if it were to be written using the current font.

See a usage *example* given along with the **PDF_SETFONT** description.

Reports

Unrelated to the family of functions dedicated to generic PDF documents production, is a function meant for the production of a whole report document.
Reports are complex aggregates of information, pre-configured in the project at design time, end produced by the runtime in form of PDF documents.

Detailed specifications about reports concepts, behaviour, configuration and compilation can be found in:

[3] PDF Reports
R:\WCE\Documenti\BlueOcean\RT7\SystemCore\ PDF Reports [x.x].docx
Description of implemented PDF Reports, concepts, configurability and compilation models

REPORT_EXPORT

Create a whole report in a PDF document.

REPORT_EXPORT (REPORTID, FILENAME, UTC, SIGNATURE [, TIMEFROM [, TIMETO [, TIMESPAN [, LANGUAGE]]]])

input			
REPORTID :	UDINT		ID of the required report
FILENAME :	ANY_STRING		name of the newly created PDF file
UTC :	BOOL		states whether the times provided (either here or in the report configuration) and the times produced (printed in the document) have to be interpreted and formatted in UTC (TRUE) or in server local time (FALSE)
SIGNATURE :	ANY_STRING		if provided, it's the signature entered by the user when the report production has been requested; can be used to force a specific signature in the dedicated report fields, if the report itself is designed to print it; can be an empty string if not needed
TIMEFROM :	DT	[OPTIONAL]	can be given to force a specific initial timestamp for the time-ranges of all the report modules that support it; if not given (along with the subsequent optional parameters) it means the original pre-configured "from" report properties won't have to be overwritten; **0xFFFFFFFF** = CDT(16#FFFFFFFF) can be given to explicitly state that this TIMEFROM must not overwrite the report properties
TIMETO :	DT	[OPTIONAL]	can be given to force a specific final timestamp for the time-ranges of all the report modules that support it; if not given (along with the subsequent optional parameters) it means the original pre-configured "to" report properties won't have to be overwritten; **0x00000000** = CDT(0) can be given to explicitly state that this TIMETO must not overwrite the report properties
TIMESPAN :	ANY_INT	[OPTIONAL]	can be given to force a specific total span for the time-ranges of all the report modules that support it; if not given it means the original pre-configured "span" report properties won't have to be overwritten; **0** can be given to explicitly state that this TIMESPAN must not overwrite the report properties

LANGUAGE :	ANY_INT	[OPTIONAL]	can be given to force a specific language to be used in multilanguage elements of the report (tables headers, text labels…); if not given (or explicitly given = **0**) it means that the system should use the language currently active in the machine (either client or server) that invoked the script execution

REPORT_ENABLESECTION

Enables or disables single modules or whole classes of sections within report documents.

REPORT_ENABLESECTION (MODULEID, ENABLE)

input

MODULEID :	ANY_INT	provides the identification of the module (or modules) that has to be enabled or disabled; allowed values are the following: - can be **0** (zero) to state that all the modules have to be enabled or disabled at once; beware: with this selection both the CLASS-oriented and the MODULE-oriented selections are affected; - can be a **positive** value to select the ID of the class of modules that must be enabled or disabled (only for CLASS-oriented selections); the supported class codes are: **1** (REPORTCLASSFDA) all FDA modules **2** (REPORTCLASSACTIVE) all ACTIVE alarms modules **3** (REPORTCLASSHISTORY) all alarms HISTORY modules **4** (REPORTCLASSSTATS) all alarms STATS modules **5** (REPORTCLASSRECIPES) all RECIPES modules **6** (REPORTCLASSDATALOG) all DATALOG modules **7** (REPORTCLASSTREND) all TREND modules **8** (REPORTCLASSCSV) all CSV modules **9** (REPORTCLASSTAGS) all TAGS list modules **10** (REPORTCLASSCUSTOM) all CUSTOM AREA modules **11** (REPORTCLASSUSERS) all FDA/USERS modules - can be a **negative** value to select the index (base-1) of the report section that must be enabled or disabled (only for MODULE-oriented selection); in this case **-1** identifies the 1st section, **-2** identifies the 2nd section, and so on
ENABLE :	BOOL	TRUE to enable the sections of the given class; FALSE to disable them

At startup all the sections are enabled by default.
Users are allowed to disable or enable
. all the modules at once,
. single modules (by selecting their index),
. or entire classes of modules (by selecting the classes they need).

Changes applied with this function will take effect starting from the next created report documents, and will remain in effect until modified or revoked. Selections must be done with this in mind, since (especially in case of multiple different reports defined in a project) there is no direct connection between the given modules IDs and any specific report.

Note that the REPORTCLASSFDA selection will only enable/disable FDA modules meant for ALL the logged records, while the REPORTCLASSUSERS selection is used for FDA tables marked for USERS events only.

< VARIABLES >

The following are the variables usable to access information about PDF document elements;

- all of the listed variables will give a proper value only while a PDF document is being produced (that is between a PDF_OPEN and a PDF_CLOSE call); their initial value is assigned as soon as a document is created; if no document is under construction, then only 0s will be found;
- all of the returned measures are given in 72nd of inch (fractional measures are possible as well);
- negative values mean there were issues trying to retrieve the measure.

PDF_PAGEWIDTH
type LREAL
access R
gives the width of the PDF document pages

PDF_PAGEHEIGHT
type LREAL
access R
gives the height of the PDF document pages

PDF_FONTHEIGHT
type LREAL
access R
gives the total height of the font currently selected in the PDF document (size of texts written with the current font);
the value of this variable is updated every time a new font is selected, every time a new font size is selected, and at the moment of the creation of the document (when the default font is enabled); see PDF_OPEN, PDF_SETFONT, PDF_SETFONTNAME and PDF_SETFONTSIZE

PDF_FONTASCENT
type LREAL
access R
gives the height of the 'ascent' portion of the font currently selected in the PDF document (see metrics in the picture below);
the value of this variable is updated every time a new font is selected, every time a new font size is selected, and at the moment of the creation of the document (when the default font is enabled); see PDF_OPEN, PDF_SETFONT, PDF_SETFONTNAME and PDF_SETFONTSIZE

PDF_FONTDESCENT
type LREAL
access R
gives the height of the 'descent' portion of the font currently selected in the PDF document (see metrics in the picture below);
the value of this variable is updated every time a new font is selected, every time a new font size is selected, and at the moment of the creation of the document (when the default font is enabled); see PDF_OPEN, PDF_SETFONT, PDF_SETFONTNAME and PDF_SETFONTSIZE

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
PDFCTXPRINT	0x0004	PDF_OPEN
PDFCTXEDIT	0x0008	PDF_OPEN
PDFCTXCOPY	0x0010	PDF_OPEN
PDFCTXEDITNOTES	0x0020	PDF_OPEN
PDFCTXFILLANDSIGN	0x0100	PDF_OPEN
PDFCTXACCESSIBLE	0x0200	PDF_OPEN
PDFCTXDOCASSEMBLY	0x0400	PDF_OPEN
PDFCTXHIGHPRINT	0x0800	PDF_OPEN
PDFCTXALL	0xFFFF	PDF_OPEN
REPORTCLASSFDA	1	REPORT_ENABLESECTION
REPORTCLASSACTIVE	2	REPORT_ENABLESECTION
REPORTCLASSHISTORY	3	REPORT_ENABLESECTION
REPORTCLASSSTATS	4	REPORT_ENABLESECTION
REPORTCLASSRECIPES	5	REPORT_ENABLESECTION
REPORTCLASSDATALOG	6	REPORT_ENABLESECTION
REPORTCLASSTREND	7	REPORT_ENABLESECTION
REPORTCLASSCSV	8	REPORT_ENABLESECTION
REPORTCLASSTAGS	9	REPORT_ENABLESECTION
REPORTCLASSCUSTOM	10	REPORT_ENABLESECTION
REPORTCLASSUSERS	11	REPORT_ENABLESECTION

10. COMMON - EXTERNAL

EveryWare

EW_ON

Starts the EveryWare client process.

EW_ON ()

EW_OFF

Stops the EveryWare client process.

EW_OFF ()

EW_ENABLE

Enables EveryWare connection and functionalities.

EW_ENABLE ()

EW_DISABLE

Disables EveryWare connection and functionalities.

EW_DISABLE ()

EW_STATE

Retrieves the current EveryWare working state.

STATE = **EW_STATE** ()

output

STATE :	UDINT	a code giving the current EveryWare state; possible codes are: **0** : EveryWare process doesn't exist **1** : EveryWare is loaded, enabled and working **2** : EveryWare is loaded, but is currently disabled **0xFFFFFFFF** : no valid state could be retrieved (can't reach the process) **N** (anything else) : EveryWare is loaded, but connection attempts ended up in failure with this error code

EW_EXIST

Checks whether the EveryWare client process is currently in execution.

STATE = **EW_EXIST** ()

output		
STATE :	BOOL	a code giving the current EveryWare process state; possible codes are: **TRUE** : EveryWare process is loaded **FALSE** : EveryWare process is NOT loaded

Messaging

MSG_EMAIL

Sends an e-mail to a list of given recipients.

MSG_EMAIL (ADDRESSES, ADDCOPY, ADDHIDDEN, REPLYTO, SUBJECT, MESSAGE, ATTACHMENTS)

input

ADDRESSES :	ANY_STRING	e-mail address of the recipient(s); in case of multiple recipients, a whole list of addresses can be given as a TAB-separated list of string pieces
ADDCOPY :	ANY_STRING	e-mail address of the recipient(s) to be set in copy; in case of multiple recipients, a whole list of addresses can be given as a TAB-separated list of string pieces; can be an empty string if not needed
ADDHIDDEN :	ANY_STRING	e-mail address of the recipient(s) to be set as hidden; in case of multiple recipients, a whole list of addresses can be given as a TAB-separated list of string pieces; can be an empty string if not needed
REPLYTO :	ANY_STRING	e-mail address to be used by recipients in Reply-To messages; can be an empty string if not needed
SUBJECT :	ANY_STRING	the subject of the e-mail; can be an empty string if not needed
MESSAGE :	ANY_STRING	the content of the e-mail; can be an empty string if not needed
ATTACHMENTS :	ANY_STRING	the file name of an attachment; multiple attachments can be given as a TAB-separated list of file names; can be an empty string if not needed

MSG_EMAILLIST

Sends an e-mail to all the users listed as members of a pre-configured mailing list.

MSG_EMAILLIST (MAILLIST, REPLYTO, SUBJECT, MESSAGE, ATTACHMENTS)

input		
MAILLIST :	ANY_STRING	name of the users mailing list; the list gathers all the recipients addresses, along with their participation modes, as normal/copy/hidden members
REPLYTO :	ANY_STRING	e-mail address to be used by recipients in Reply-To messages; can be an empty string if not needed
SUBJECT :	ANY_STRING	the subject of the e-mail; can be an empty string if not needed
MESSAGE :	ANY_STRING	the content of the e-mail; can be an empty string if not needed
ATTACHMENTS :	ANY_STRING	the file name of an attachment; multiple attachments can be given as a TAB-separated list of file names; can be an empty string if not needed

MSG_APPNOTIFICATION

Sends an app notification to a list of given recipients.

MSG_APPNOTIFICATION (ADDRESSES, MESSAGE)

input		
ADDRESSES :	ANY_STRING	e-mail address of the recipient(s); in case of multiple recipients, a whole list of addresses can be given as a TAB-separated list of string pieces
MESSAGE :	ANY_STRING	the content of the notification message; can be an empty string if not needed

MSG_APPNOTIFICATIONLIST

Sends an app notification to all the users listed as members of a pre-configured mailing list.

MSG_APPNOTIFICATIONLIST (MAILLIST, MESSAGE)

input		
MAILLIST :	ANY_STRING	name of the users mailing list; the list gathers all the recipients addresses
MESSAGE :	ANY_STRING	the content of the notification message; can be an empty string if not needed

MSG_SMS

Sends an SMS to a list of given recipients.

MSG_SMS (NUMBERS, MESSAGE)

input

NUMBERS :	ANY_STRING	telephone (mobile) number of the recipient(s); in case of multiple recipients, a whole list of numbers can be given as a TAB-separated list of string pieces
MESSAGE :	ANY_STRING	the content of the SMS message; can be an empty string if not needed

MSG_SMSLIST

Sends an SMS to all the users listed as members of a pre-configured mailing list.

MSG_SMSLIST (MAILLIST, MESSAGE)

input		
MAILLIST :	ANY_STRING	name of the users mailing list; the list gathers all the recipients' telephone (mobile) numbers
MESSAGE :	ANY_STRING	the content of the SMS message; can be an empty string if not needed

11. RUNTIME - TIMERS

TIMER_START

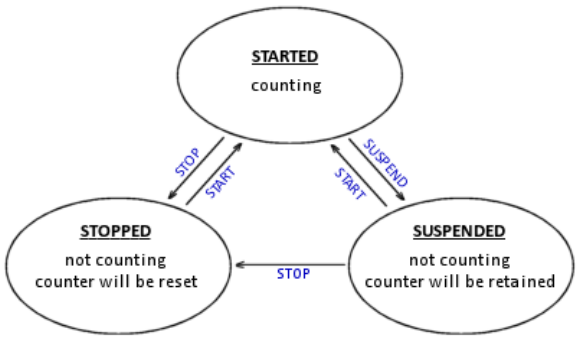
Starts a timer counting and its ability to generate 'fire' events.

TIMER_START (TIMER)

input

TIMER :	ANY_STRING	the name of the timer

If the timer is already started, this instruction is ignored;
if it's currently stopped, its progress counter is reset before the start;
if it's currently suspended, its progress counter is retained, and the counting goes on from there.



TIMER_STOP

Stops a timer counting and its ability to generate 'fire' events.

TIMER_STOP (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

If the timer is already stopped, this instruction is ignored;
if it's currently suspended, the state becomes 'stopped'
(suspended timers can be downgraded to stopped, but not the opposite);
if it's currently started, the timer is just stopped.

While stopped, the timer retains its progress counter value as it is; when restarted though, the counter is reset so that a whole new period is counted.
See **TIMER_START** for a description of the states changes.

TIMER_SUSPEND

Suspends a timer counting and its ability to generate 'fire' events.

TIMER_SUSPEND (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

If the timer is already suspended or stopped (not counting), this instruction is ignored
(only started timers can be suspended);
if it's currently started (counting), the counting is stopped and the timer state becomes suspended.

While suspended, the timer retains its progress counter value as it is; if restarted, the count goes on from there.
See **TIMER_START** for a description of the states changes.

TIMER_SETLIMIT

Sets a new value for the timer counter limit.

TIMER_SETLIMIT (TIMER, LIMIT)

input		
TIMER :	ANY_STRING	the name of the timer
LIMIT :	ANY_ELEMENTARY	the new limit value this parameter can actually be of the following types only: TIME, LTIME, DT, LDT, TOD, LTOD, ANY_UNSIGNED (see below for a description of supported modes and types)

Values and types of the LIMIT parameter actually depend on the type of the timer, since different timers are designed to count different things. The timer type must be known by the programmer: it's coming from the project configuration and can't change at runtime.

ONESHOT and **NORMAL** timers (the two period-based timers) need a limit that gives the duration of their period;
this limit is expected to have a resolution in 10^{ths} of second;
it can be given here in the following forms:
- TIME : the TIME duration, natively in ms, is cut to the lower 10^{th} of second;
- LTIME : the LTIME duration, natively in ns, is cut to the lower 10^{th} of second;
- ANY_UNSIGNED : the given value must be a time duration expressed in 10^{ths} of second.

ALARMSINGLE needs a limit giving the date and time of its one and only fire event (local server time);
the expected precision for the event is in seconds (precision and range of a DT are enough);
this limit can be given here in the following forms:
- DT : the DT (simple date and time) natively gives the date and time with the correct specs;
- LDT : the LDT (long date and time) is cut in precision and range to those of a DT;
- ANY_UNSIGNED : the given value must be exactly the numeric value of a corresponding DT: that means the number of seconds elapsed since the 00:00 of 1.1.1970 (the numeric value of a standard UNIX time).

ALARMTIME needs a limit giving the time of its daily events (local server time);
the expected precision for the event is in seconds;
this limit can be given here in the following forms:
- TOD : the TOD value, natively in ms, is cut to the lower second;
- LTOD : the LTOD value, natively in ns, is cut to the lower second;
- ANY_UNSIGNED : the given value must be a time expressed in number of seconds elapsed since the 00:00.

TIMER_GETLIMIT

Obtains the current value of the timer counter limit.

LIMIT = **TIMER_GETLIMIT** (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

output		
LIMIT :	ANY_ELEMENTARY	the current timer limit value this result can actually be of the following types only: LTIME, DT, TOD (see below for a description of supported modes and types)

Values and types of the LIMIT result actually depend on the type of the timer, since different timers are designed to count different things. The timer type must be known by the programmer: it's coming from the project configuration and can't change at runtime.

ONESHOT and **NORMAL** timers (the two period-based timers) have a limit that gives the duration of their period; it's returned by this function as a:
- LTIME : the returned value will be multiple of the counter resolution (10^{ths} of second).

ALARMSINGLE has a limit giving the date and time of its one and only fire event (local server time); it's returned by this function as a:
- DT : the returned DT matches the needed specs for the limit range and resolution (seconds).

ALARMTIME has a limit giving the time of its daily events (local server time); it's returned by this function as a:
- TOD : the returned TOD will be a multiple of the counter resolution (seconds).

TIMER_SETPROGRESS

Sets a new value for the timer progress counter.

TIMER_SETPROGRESS (TIMER, PROGRESS)

input

TIMER :	ANY_STRING	the name of the timer
PROGRESS :	ANY_MAGNITUDE	the new counter progress value; this parameter can actually be of the following types only: TIME, LTIME, ANY_UNSIGNED (see below for a description of supported types)

The PROGRESS parameter is given as the new value for the timer progress counter: the counter that states the amount of time missing to the timer fire event.
This is essentially a time duration; its needed precision is in 10^{ths} of second.
It can be given in the following forms:
- TIME : the given duration, natively in ms, is cut to the lower 10^{th} of second;
- LTIME : the given duration, natively in ns, is cut to the lower 10^{th} of second;
- ANY_UNSIGNED : the given value must be a time duration expressed in 10^{ths} of second.

TIMER_GETPROGRESS

Obtains the current value of the timer progress counter.

PROGRESS = **TIMER_GETPROGRESS** (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

output		
PROGRESS :	LTIME	the current counter progress value

This is the counter that states the amount of time missing to the timer fire event.
The counter resolution is in 10^{ths} of second.
The returned value though, multiple of these 10^{ths} of second, is packed in an LTIME.

TIMER_ISSTARTED

Checks whether a timer is currently counting or not.

STATE = **TIMER_ISSTARTED** (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

output		
STATE :	BOOL	TRUE if the timer is currently started; FALSE if it's currently stopped or suspended

TIMER_ISSUSPENDED

Checks whether a timer is currently suspended or not.

STATE = **TIMER_ISSUSPENDED** (TIMER)

input		
TIMER :	ANY_STRING	the name of the timer

output		
STATE :	BOOL	TRUE if the timer is currently suspended; FALSE if it's currently started or stopped

12. RUNTIME - TAGS

TAG_GETVALUE

Retrieves the current tag value.

VALUE = **TAG_GETVALUE** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
VALUE :	ANY	the value of the tag; the type of the returned value will match the (internal) type of the given tag

This function doesn't cause any acquisition from a device: the returned value is directly taken from the current tag cache.

TAG_SETVALUE

Changes the current tag value.
The value is kept within the tag, not written to the device.

TAG_SETVALUE (TAG, VALUE)

input

TAG :	ANY_STRING	the name of the tag
VALUE :	ANY	the new value for the tag; the type of the value should match the (internal) type of the given tag; types conversions are actually automatically handled to a certain degree, but coherence should be granted by the programmer (see **TAG_WRITEVALUE** for hints)

This function is meant to store the new value in the tag internal memory, while avoiding writing it on the device.
When a value set in the tag has to be finalized on the device, an explicit call to a **TAG_FLUSHVALUE** is necessary.

TAG_FLUSHVALUE

Writes on device the value currently stored in a tag.

TAG_FLUSHVALUE (TAG)

input

TAG :	ANY_STRING	the name of the tag

This function expects the tag to already have a valid value in its internal memory, prepared by previous acquisitions or by application assignments. Especially meant to be used after values have been assigned by functions like **TAG_SETVALUE** or **TAG_SETFIELDVALUE**.

TAG_READVALUE

Reads from device the value of a tag.

VALUE = **TAG_READVALUE** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
VALUE :	ANY	the value of the tag; the type of the returned value will match the (internal) type of the given tag

The value of the tag is acquired from its device, stored in the tag memory, and then returned.

This function works synchronously and is blocking for the script (by the time the function exits, the communication with the device is complete and the tag is already updated).

TAG_WRITEVALUE

Writes on device the value of a tag.

TAG_WRITEVALUE (TAG, VALUE)

input

TAG :	ANY_STRING	the name of the tag
VALUE :	ANY	the value of the tag; the type of the value should match the (internal) type of the given tag; types conversions are actually automatically handled to a certain degree, but coherence should be granted by the programmer; for example, the system is normally able to convert any kind of plain numeric value in case of plain numeric tag, but conversions between strings and numbers are not supported; also particular attention must be paid in case of arrays, since the system only allows the usage of precisely matching types

The value of the tag is stored in the tag memory, and then written on its device.

This function works synchronously and is blocking for the script (by the time the function exits, even the communication with the device is complete).

TAG_READELEMENT

Reads from device the value of the element of a tag-array.

VALUE = **TAG_READELEMENT** (TAG, INDEX)

input		
TAG :	ANY_STRING	the name of the tag
INDEX :	ANY_INT	the index (base-0) of the array element

output		
VALUE :	ANY	the value of the tag-array element; the type of the returned value will match the (internal) type of the given element

The operation actually affects the whole tag: the whole array is acquired from the device and stored in the tag memory; then the value of the single element is extracted from the tag and returned.

This function works synchronously and is blocking for the script (by the time the function exits, the communication with the device is complete and the tag is already updated).

TAG_WRITEELEMENT

Writes on device the value of the element of a tag-array.

TAG_WRITEELEMENT (TAG, INDEX, VALUE)

input

TAG :	ANY_STRING	the name of the tag
INDEX :	ANY_INT	the index (base-0) of the array element
VALUE :	ANY	the value of the tag element; the type of the value should match the (internal) type of the given element types conversions are actually automatically handled to a certain degree, but coherence should be granted by the programmer

The operation actually affects the whole tag: the given element value is used to update the tag-array value in memory; then the whole array value is written on the device.

This function works synchronously and is blocking for the script (by the time the function exits, even the communication with the device is complete).

TAG_READBIT

Reads from device the value of the bit of a numeric tag.

VALUE = **TAG_READBIT** (TAG, INDEX)

input		
TAG :	ANY_STRING	the name of the tag
INDEX :	ANY_INT	the index (base-0) of the bit

output		
VALUE :	BOOL	the value of the tag bit

The operation is only supported on integer numeric tags (booleans included), and on corresponding arrays of numeric elements. Essentially on everything that doesn't involve values of string and floating-point types.

The operation actually affects the whole tag: it is acquired as a whole from the device and stored in the tag memory; then the value of the single bit is extracted from the tag and returned.

This function works synchronously and is blocking for the script (by the time the function exits, the communication with the device is complete and the tag is already updated).

TAG_WRITEBIT

Writes on device the value of the bit of a numeric tag.

TAG_WRITEBIT (TAG, INDEX, VALUE)

input

TAG :	ANY_STRING	the name of the tag
INDEX :	ANY_INT	the index (base-0) of the bit
VALUE :	BOOL	the new value of the tag bit

The operation is only supported on integer numeric tags (booleans included), and on corresponding arrays of numeric elements. Essentially on everything that doesn't involve values of string and floating-point types.

The operation actually affects the whole tag: the given bit value is used to update the tag value in memory; then the whole tag value is written on the device.

This function works synchronously and is blocking for the script (by the time the function exits, even the communication with the device is complete).

TAG_READITEM

Reads from device any addressable element.

VALUE = **TAG_READITEM** (DEVICE, AREA, TYPE, STRLEN, ASIZE, BCD, ADD1 [, ADD2 [, ADD3 [, ... , ADD8]]])

input

DEVICE :	ANY_INT	the ID of the device where the tag is mapped; this code depends on the configuration of the current project; see TAG_GETDEVICEID for a way to retrieve the device ID of a known tag; see TAG_DEVICESNUMBER, TAG_DEVICEGETID and TAG_DEVICEGETNAME for a way to browse the devices existing in the project
AREA :	ANY_INT	the ID of the area within the given device memory; this code depends on the implementation of the communication device in use; see TAG_GETAREAID for a way to retrieve the device area ID of a known tag; see TAG_AREASNUMBER, TAG_AREAGETID and TAG_AREAGETNAME for a way to browse the memory areas supported by a given device
TYPE :	ANY_INT	the code of the type of the value that has to be acquired; the supported codes (along with their symbolic constants) are: - 16 (TAGTYPESINT) - 17 (TAGTYPEUSINT) - 2 (TAGTYPEINT) - 18 (TAGTYPEUINT) - 3 (TAGTYPEDINT) - 19 (TAGTYPEUDINT) - 20 (TAGTYPELINT) - 21 (TAGTYPEULINT) - 4 (TAGTYPEREAL) - 5 (TAGTYPELREAL) - 11 (TAGTYPEBOOL) - 30 (TAGTYPESTRING) - 8 (TAGTYPEWSTRING) in case of array, the following constant must be added to the element type: - 0x2000 (TAGARRAY)
STRLEN :	ANY_INT	the length of the string, if the requested item type is string; it's ignored otherwise
ASIZE :	ANY_INT	the size (number of elements) of the array, if the requested item is an array; it's ignored otherwise
BCD :	BOOL	TRUE if the item value is in BCD on the device; FALSE otherwise
ADD# :	ANY	a list of (up to) 8 parameters, giving the values of the address fields of the required item (the address of the starting location of the item in the device memory); the number of parameters ranges between 1 and 8; their actual number and types depend on the device and the referenced memory area; the programmer is expected to know the needs of the targeted memory area; hints about it can be deduced from the tags configuration windows in Crew™

output

VALUE :	ANY	the value of the acquired item; expected to match the TYPE given as parameter

This function works synchronously and is blocking for the script (by the time the function exits, the whole communication process with the device is complete).

A special note about "variant" accesses: in case of items that can be identified as children of predefined structured tags, it is possible to submit special requests, where:
- DEVICE = 0
- AREA = 0
- TYPE = 0
- one only ADD# field, formatted as
  ADD1 = field identification = "TAG.path...path"

In this case, with most of the properties empty, the address field (formatted as a concatenation of a tag name and a structured path leading to the targeted field) is used to identify a tag and a structured field; most of the item properties are automatically identified by the runtime, in part inherited from the tag, in part found in the (already known) structure definition.

TAG_WRITEITEM

Writes on device any addressable element.

TAG_WRITEITEM (VALUE, DEVICE, AREA, TYPE, STRLEN, ASIZE, BCD, ADD1 [, ADD2 [, ADD3 [, … , ADD8]]])

input

VALUE :	ANY	the value of the item; expected to match the TYPE specified along (a certain degree of types compatibility is allowed by the system; see **TAG_WRITEVALUE** for hints on the matter)
DEVICE :	ANY_INT	the ID of the device where the tag is mapped
AREA :	ANY_INT	the ID of the area within the given device memory
TYPE :	ANY_INT	the code of the type of the value that has to be acquired
STRLEN :	ANY_INT	the length of the string, if the requested item type is string
ASIZE :	ANY_INT	the size (number of elements) of the array, if the requested item is an array
BCD :	BOOL	TRUE if the item value is in BCD on the device; FALSE otherwise
ADD# :	ANY	a list of (up to) 8 parameters, giving the values of the address fields of the required item (the address of the starting location of the item in the device memory)

See **TAG_READITEM** for deeper and detailed specifications of meanings and mechanics involved with the given parameters.

As above (like **TAG_READITEM**) this function supports "variant" accesses as well: in case of items that can be identified as children of predefined structured tags, it is possible to submit special requests, where:
- DEVICE = 0
- AREA = 0
- TYPE = 0
- one only ADD# field, formatted as
  ADD1 = field identification = "TAG.path…path"

The address field (formatted as a concatenation of a tag name and a structured path leading to the targeted field) is used to identify a tag and one of its fields; most of the item properties are then automatically identified by the runtime, in part inherited from the tag, in part found in the (already known) structure definition.

This function works synchronously and is blocking for the script (by the time the function exits, the whole communication process with the device is complete).

example

```
VAR
   idx : UINT;
END_VAR;

// …how to write 100 items on a modbus…
FOR idx := 0 TO 99 DO
   TAG_WRITEITEM ( idx,                  // write the progress counter
                   1,                    // let's say the needed device is the 1st of the project
                   2,                    // regular modbus registers are on area n. 2
                   TAGTYPEUINT,          // same as <idx> type; matching the addressable modbus unit
                   0,                    // not a string
                   0,                    // not an array
                   FALSE,                // not BCD
                   ANY_TO_UDINT(idx) );  // register address, needed as UDINT
END_FOR;
```

TAG_GETID

Retrieves the ID of a tag of given name.

ID = **TAG_GETID** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
ID :	UDINT	the ID of the tag

TAG_GETNAME

Retrieves the name of a tag of given ID.

```
TAG = TAG_GETNAME (ID)
```

input		
ID :	UDINT	the ID of the tag

output		
TAG :	WSTRING	the name of the tag

TAG_GETSHAREDID

Retrieves the shared ID (public network ID) of a tag of given name.

SHARED = **TAG_GETSHAREDID** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
SHARED :	UDINT	the shared ID of the tag

TAG_GETIDFROMSHARED

Retrieves the ID of a tag of given shared ID.

ID = **TAG_GETIDFROMSHARED** (SHARED)

input		
SHARED :	UDINT	the shared ID of the tag

output		
ID :	UDINT	the ID of the tag

TAG_GETVALUETYPE

Retrieves the type code of a tag value.

TYPE = **TAG_GETVALUETYPE** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
TYPE :	UINT	the code of the tag value type; the supported codes (along with their symbolic constants) are: - 16 (**TAGTYPESINT**) 8 bits signed integer - 17 (**TAGTYPEUSINT**) 8 bits unsigned integer - 2 (**TAGTYPEINT**) 16 bits signed integer - 18 (**TAGTYPEUINT**) 16 bits unsigned integer - 3 (**TAGTYPEDINT**) 32 bits signed integer - 19 (**TAGTYPEUDINT**) 32 bits unsigned integer - 20 (**TAGTYPELINT**) 64 bits signed integer - 21 (**TAGTYPEULINT**) 64 bits unsigned integer - 4 (**TAGTYPEREAL**) 32 bits floating-point - 5 (**TAGTYPELREAL**) 64 bits floating-point - 11 (**TAGTYPEBOOL**) boolean - 30 (**TAGTYPESTRING**) 8 bits chars string - 8 (**TAGTYPEWSTRING**) 16 bits chars string in case of array, the following constant is added to the element type: - 0x2000 (**TAGARRAY**)

TAG_GETSTRLENGTH

Retrieves the length of the string value of a tag.

LENGTH = **TAG_GETSTRLENGTH** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
LENGTH :	UINT	the length of the string value (in case of string value type); 0 otherwise

If the given tag has not a string type, this method returns 0; programmers should not rely on this information to check the type of the tag value though: see **TAG_GETVALUETYPE** instead.

TAG_GETARRAYSIZE

Retrieves the size (number of elements) of a tag-array.

SIZE = **TAG_GETARRAYSIZE** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
SIZE :	UINT	the number of elements of the tag-array; 0 if it's not an array

If the given tag is not an array, this method returns 0; programmers should not rely on this information to check the type of the tag value though: see **TAG_GETVALUETYPE** instead.

TAG_GETDEVICEID

Retrieves the ID of the device where a given tag is located.

ID = **TAG_GETDEVICEID** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
ID :	UINT	the ID of the tag's device; these IDs are defined as indexes (base-1) of the device within the project devices collection; see **TAG_DEVICESNUMBER**, **TAG_DEVICEGETID** and **TAG_DEVICEGETNAME** for tools usable to browse the existing devices

In case of variable addresses, the function returns the current device ID value.

TAG_GETAREAID

Retrieves the ID of the device memory area where a given tag is mapped.

ID = **TAG_GETAREAID** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
ID :	UINT	the ID of the tag's device memory area; these IDs are defined as indexes (base-1) of the area within the tag's device memory areas collection; see **TAG_AREASNUMBER**, **TAG_AREAGETID** and **TAG_AREAGETNAME** for tools usable to browse the existing areas

In case of variable addresses, the function returns the current area ID value.

TAG_GETADDRESS

Retrieves the value of an address field of a given tag.

ADDRESS = **TAG_GETADDRESS** (TAG, INDEX)

input		
TAG :	ANY_STRING	the name of the tag
INDEX :	ANY_INT	the index of the needed address field; being the address of a tag defined by a number of fields (number and types depending on the tag's device and memory area), this parameter gives the index (base-0) of the field within the whole address fields sequence; the range of allowed indexes (the composition of the complete address) is expected to be known by the programmer

output		
ADDRESS :	ANY	the value of the required address field

In case of variable addresses, the function returns the current address field value.

TAG_GETFIELDOFFSET

Retrieves the offset of a structure field within a given structured tag.

OFFSET = **TAG_GETFIELDOFFSET** (TAG, PATH)

input		
TAG :	ANY_STRING	the name of the tag
PATH :	ANY_STRING	the path that identifies the structure field; must correspond to the layout of the given tag structured type; must start with either a '[' (in case the tag is an array and a specific element has to be accessed) or a '.' (a dot, for plain field access); in principle, the exact concatenation of TAG and PATH must give a complete structured identification of the field
output		
OFFSET :	UDINT	the offset, in bytes, of the beginning of the field value within the complete structured value of the tag

The method is designed to work with internal structured tags as well as with any structured tag with numeric address, provided they have been precisely defined in the declaration of their type (among the information compiled for the server runtime).
In case of tags with symbolic address, the fields addresses are given by plain concatenations of the involved symbolic strings, and are not taken into account by this function.

TAG_GETFIELDADDRESS

Retrieves the address of a structure field of a given structured tag.

ADDRESS = **TAG_GETFIELDADDRESS** (TAG, PATH)

input		
TAG :	ANY_STRING	the name of the tag
PATH :	ANY_STRING	the path that identifies the structure field; must correspond to the layout of the given tag structured type; must start with either a '[' (in case the tag is an array and a specific element has to be accessed) or a '.' (a dot, for plain field access); in principle, the exact concatenation of TAG and PATH must give a complete structured identification of the field
output		
ADDRESS :	UDINT	the address of the targeted structure field, calculated adding its internal offset to the base address of the tag

The method is designed to work with internal structured tags as well as with any structured tag with numeric "byte" address, provided they have been precisely defined in the declaration of their type (among the information compiled for the server runtime).
In case of tags with symbolic address, the fields addresses are given by plain concatenations of the involved symbolic strings, and are not taken into account by this function.

Note the "byte" in the statement above: the result is calculated adding a number of bytes to a base address, so a "byte addressing" is the intended model for the given tag.
A second limitation: in case of addresses made by multiple fields, the offset is added to the value of the last address field of the base tag address, and only that address field result is returned.
In other words: this function works well with internal tags (mainly the tags for which it has been designed) and in general with tags
- with byte numeric addressing and
- with one only address field, or tags for which it's safe to assume that the offset component can be added to the last address field.

In cases where the limitations above are not met, this function should be avoided, and a combination of **TAG_GETADDRESS** and **TAG_GETFIELDOFFSET** along with appropriate calculations should be used instead.

TAG_GETFIELDVALUE

Retrieves the value of a single structure field within a given structured tag.

VALUE = **TAG_GETFIELDVALUE** (TAG, PATH)

input		
TAG :	ANY_STRING	the name of the 'parent' structured tag
PATH :	ANY_STRING	the path that identifies the field; must correspond to the layout of the given tag structured type; must start with either a '[' (in case the tag is an array and a specific element has to be accessed) or a '.' (a dot, for plain field access); in principle, the exact concatenation of TAG and PATH must give a complete structured identification of the field

output		
VALUE :	ANY	the value of the field; the type of the returned value will match the type of the given field, as defined in the involved structure

The function expects the given structure tag to already have a value, either acquired from the field, or assigned by some application logic. The returned field value is not acquired from the field device: it is instead simply extracted from the value already owned by its parent structured tag.

TAG_SETFIELDVALUE

Changes the value of a single structure field within a given structured tag.

TAG_SETFIELDVALUE (TAG, PATH, VALUE)

input		
TAG :	ANY_STRING	the name of the 'parent' structured tag
PATH :	ANY_STRING	the path that identifies the field; must correspond to the layout of the given tag structured type; must start with either a '[' (in case the tag is an array and a specific element has to be accessed) or a '.' (a dot, for plain field access); in principle, the exact concatenation of TAG and PATH must give a complete structured identification of the field
VALUE :	ANY	the value of the field; the type of the returned value MUST match the type of the given field, as defined in the involved structure

The function expects the given structure tag to already have a value, either acquired from the field, or assigned by some application logic. The function will simply change a part of the whole value.
Note that the given value is retained in the tag internal memory, but not written to the device; when a value set in the tag has to be finalized on the device, an explicit call to a **TAG_FLUSHVALUE** is necessary.

TAG_ASSIGNSTRUCT

Assign to all the fields of an ST (user-defined) structured value, the values coming from a matching structured tag.

VALUE = **TAG_ASSIGNSTRUCT** (TAG, TYPE)

input		
TAG :	ANY_STRING	the name of the source structured tag
TYPE :	ANY_STRING	the name of an ST user-defined structure type; only structures and arrays of structures are acceptable types

output		
VALUE :	ANY	output value; this value, built as function return, assumes the exact type of the structure (or array of structures) identified by the name given in the TYPE

The function is meant to assign fields values, from a structure to another, in cases where the structures are not perfectly identical.
The source (input TAG) is a structured tag, the destination (output VALUE) is an ST value of a structured type.

Assignment rules:
- assignments are done field by field;
- fields match is done by name;
- fields that exist in the source but not in the destination, are ignored;
- fields that exist in the destination but not in the source, are explicitly reset;
- fields types should match; perfection is not required, but checks are strict:
  . integers can only be assigned to integers (or pseudo-integers) of the same size
  (pseudo-integers definition includes integers, ranges, enumeratives, bitstrings, booleans, date/time),
  . reals can only be assigned to reals of the same size,
  . strings can only be assigned to strings of the same size and type,
  . arrays can only be assigned to arrays of the same size and with elements following the rules above,
  . sub-structures and arrays of sub-structures are handled recursively.

TAG_GETNUMBERALL

Counts the total number of tags.

```
NUMBER = TAG_GETNUMBERALL ()
```

output

NUMBER :	UDINT	the total number of tags

The function counts all the tags configured in the current project;
the count includes:
- all external, internal and system tags,
- all indirect indexed and variable tags,
- all recipe buffer tags (actually falling under the 'internal' category),
- all sub-tags implicitly created by the compiler to support direct arrays elements access.

TAG_GETNUMBEREXT

Counts the number of external tags (tags mapped on external devices).

NUMBER = **TAG_GETNUMBEREXT** ()

output		
NUMBER :	UDINT	the number of external tags

The function counts all the external tags configured in the current project;
the count includes:
- all external tags (excludes internal and system tags),
- all indirect variable tags (excludes indexed tags),
- all sub-tags created by the compiler to access elements of external tags (excludes sub-tags for internal tags).

TAG_GETCLIENTTAGNAME

Retrieves the name of a specific client system tag.
Meant to be used in scripts invoked by client requests.

TAGNAME = **TAG_GETCLIENTTAGNAME** (TAGCODE)

input

TAGCODE :	ANY_INT	a code that identifies the needed client variable; the given code can identify one of the client system tags managed by server drivers buffers; supported codes are: **0** (TAGSYSUILASTALARM): the specialization of "SYS_UILastAlarm" **1** (TAGSYSUIMAINALARM): the specialization of "SYS_UIAlarmMainMessage" **2** (TAGSYSUILANGUAGENAME): the specialization of "SYS_UILanguageName" **3** (TAGSYSUILANGUAGEID): the specialization of "SYS_UILanguageId" **4** (TAGSYSUIKEYSBUFFER): the specialization of "SYS_UIKeysBuffer"

output

TAGNAME :	WSTRING	the name of the identified tag compiled for the calling client

The function is supposed to be used within scripts invoked by clients requests.

If the script execution originated from a server event, or if the specified tag doesn't exist in the current project, then the function will return an empty string.
If instead the script execution originated from a client request and the specified tag actually exists, then the function will return the correct tag name for the calling client.

TAG_ISOFFLINE

Checks whether a tag is currently online or offline.

STATE = **TAG_ISOFFLINE** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
STATE :	BOOL	TRUE if the tag is offline; FALSE if it's online

Tags are offline if the last operation involving a read/write of their value from/to the device went wrong. When this happens their offline state is set, and their value is invalidated.

TAG_ISOFFSCAN

Checks whether a tag is currently onscan or offscan.

STATE = **TAG_ISOFFSCAN** (TAG)

input		
TAG :	ANY_STRING	the name of the tag

output		
STATE :	BOOL	TRUE if the tag is offscan; FALSE if it's onscan

Tags can be offscan as a result of a **TAG_SETOFFSCAN** or **TAG_SETOFFSCANDEV** request, or because of an initial project configuration.
Offscan tags inhibit the link between their value and the device where they are mapped.
When a value is written in an offscan tag, it is stored in the tag's memory, but not sent to the device; when a value is read from an offscan tag, the value is not acquired from the device, but taken directly from the tag's memory.

TAG_SETOFFSCAN

Changes the offscan state of a tag.

TAG_SETOFFSCAN (TAG, STATE)

input		
TAG :	ANY_STRING	the name of the tag
STATE :	BOOL	the new offscan state

TAG_SETOFFSCANDEV

Changes the offscan state of a whole device.

TAG_SETOFFSCANDEV (DEVICE, STATE)

input		
DEVICE :	ANY_STRING	the name of the device
STATE :	BOOL	the new device offscan state

The function expects as parameter the name of the involved device.
As tools to handle the devices names, it is always possible to use:
- **TAG_GETDEVICEID** to retrieve the ID of the device of a tag,
- **TAG_DEVICEGETNAME** to get the name of a device of known ID,
- **TAG_GETDEVICESNUMBER** to retrieve the range of all the possible devices IDs in the project.

TAG_DEVICESNUMBER

Retrieves the number of devices configured in the current project.

NUMBER = **TAG_DEVICESNUMBER** ()

output

NUMBER :	UINT	the number of configured devices

With this function, in combination with the devices' id/name conversion functions (**TAG_DEVICEGETID** and **TAG_DEVICEGETNAME**), programmers are able to browse all the configured devices.

example

```
VAR
   idx  : UINT;
   ndev : UINT;
   chk  : UINT;
   dname : WSTRING [128];
END_VAR;

ndev := TAG_DEVICESNUMBER ();
FOR idx := 1 TO ndev DO
   dname := TAG_DEVICEGETNAME (idx);     // get the name of the idx-th device
   chk := TAG_DEVICEGETID (dname);       // convert back: <chk> must match the original <idx>
   // do whatever needed with the obtained info
END_FOR;
```

TAG_DEVICEGETID

Retrieves the ID of a device of known name.

ID = **TAG_DEVICEGETID** (DEVICE)

input		
DEVICE :	ANY_STRING	the name of the device

output		
ID :	UINT	the ID of the device; these IDs are defined as indexes (base-1) of the device within the project devices collection

See **TAG_DEVICESNUMBER** for a usage example.

TAG_DEVICEGETNAME

Retrieves the name of a device of known ID.

DEVICE = **TAG_DEVICEGETNAME** (ID)

input		
ID :	UINT	the ID of the device; these IDs are defined as indexes (base-1) of the device within the project devices collection

output		
DEVICE :	WSTRING	the name of the device

See **TAG_DEVICESNUMBER** for a usage example.

TAG_AREASNUMBER

Retrieves the number of memory areas available in a given device.

NUMBER = **TAG_AREASNUMBER** (DEVID)

input

DEVID :	UINT	the ID of the interested device

input

NUMBER :	UINT	the number of memory areas available in the given device

With this function, in combination with the areas' id/name conversion functions (**TAG_AREAGETID** and **TAG_AREAGETNAME**), programmers are able to browse all the possible memory areas.

example

```
VAR
   idx   : UINT;
   nar   : UINT;
   chk   : UINT;
   aname : WSTRING [128];
   devid : UINT;
END_VAR;

devid := 1;                                  // scan the areas of the 1st device

nar := TAG_AREASNUMBER (devid);
FOR idx := 1 TO nar DO
   aname := TAG_AREAGETNAME (devid, idx);    // get the name of the idx-th area
   chk := TAG_AREAGETID (devid, aname);      // convert back: <chk> must match the original <idx>
   // do whatever needed with the obtained info
END_FOR;
```

TAG_AREAGETID

Retrieves the ID of a device memory area of known name.

ID = **TAG_AREAGETID** (DEVID, AREA)

input		
DEVID :	UINT	ID of the interested device
AREA :	ANY_STRING	the name of the device memory area

output		
ID :	UINT	the ID of the memory area; these IDs are defined as indexes (base-1) of the area within the tag's device memory areas collection

See **TAG_AREASNUMBER** for a usage example.

TAG_AREAGETNAME

Retrieves the name of a device memory area of known ID.

AREA = **TAG_AREAGETNAME** (DEVID, ID)

input		
DEVID :	UINT	the ID of the interested device
ID :	UINT	the ID of the device memory area; these IDs are defined as indexes (base-1) of the area within the tag's device memory areas collection

output		
AREA :	WSTRING	the name of the memory area

See **TAG_AREASNUMBER** for a usage example.

TAG_FLUSH

Flushes on persistent storage the values of all the persistent tags.

TAG_FLUSH ()

The operation affects both internal and external persistent tags (tags whose values have to be recovered and reapplied even after a system shutdown).

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
TAGTYPESINT	16	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEUSINT	17	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEINT	2	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEUINT	18	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEDINT	3	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEUDINT	19	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPELINT	20	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEULINT	21	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEREAL	4	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPELREAL	5	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEBOOL	11	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPESTRING	30	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGTYPEWSTRING	8	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGARRAY	0x2000	TAG_READITEM, TAG_WRITEITEM, TAG_GETVALUETYPE
TAGSYSUILASTALARM	0	TAG_GETCLIENTTAGNAME
TAGSYSUIMAINALARM	1	TAG_GETCLIENTTAGNAME
TAGSYSUILANGUAGENAME	2	TAG_GETCLIENTTAGNAME
TAGSYSUILANGUAGEID	3	TAG_GETCLIENTTAGNAME
TAGSYSUIKEYSBUFFER	4	TAG_GETCLIENTTAGNAME

13. RUNTIME - ALARMS

ALARM_ON

Raises an alarm condition.

INSTANCE = **ALARM_ON** (ALARM [, USER, STATION])

input			
ALARM :	ANY_STRING		name of the alarm
USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

output		
INSTANCE :	UDINT	ID of the created alarm instance

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

The INSTANCE ID can be later used in calls to **ALARM_ACKSINGLE**.

ALARM_OFF

Clears an alarm condition.

ALARM_OFF (ALARM [, USER, STATION])

input			
ALARM :	ANY_STRING		name of the alarm
USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

ALARM_ACKSINGLE

Acknowledges a single instance of an active alarm.

ALARM_ACKSINGLE (INSTANCE [, USER, STATION])

input			
INSTANCE :	ANY_INT		ID of the alarm instance
USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

The INSTANCE ID is coming from previous calls to **ALARM_ON**, or queries like **ALARM_GETINSTANCEID**.
It could also come from embedded events parameters passed directly by the runtime, in case of scripts associated to OnAlarmOn events.

ALARM_ACKINSTANCES

Acknowledges all the active instances of a given alarm.

ALARM_ACKINSTANCES (ALARM [, USER, STATION])

input

ALARM :	ANY_STRING		name of the alarm
USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

ALARM_ACKGROUP

Acknowledges all the active instances of all the alarms that are part of a given group.

ALARM_ACKGROUP (GROUP [, USER, STATION])

input			
GROUP :	ANY_STRING		name of the alarms group
USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

ALARM_ACKALL

Acknowledge all the active alarm instances.

ALARM_ACKALL ([USER, STATION])

input

USER :	ANY_STRING	[OPTIONAL]	name of the responsible user; if missing, automatically assumed to be the user currently logged in at server level
STATION :	ANY_STRING	[OPTIONAL]	name of the alarm source machine; if missing, automatically assumed to be the server machine

USER and STATION can't be given individually: the two optional parameters must be both specified, or both omitted.

ALARM_ISON

Checks whether a given alarm condition is currently raised (if an active instance exists for the given alarm).

STATE = **ALARM_ISON** (ALARM)

input		
ALARM :	ANY_STRING	name of the alarm

output		
STATE :	BOOL	TRUE if the alarm is active (an instance exists); FALSE otherwise

ALARM_HISTORYRESET

Removes all the logged records from the alarms history.

ALARM_HISTORYRESET ()

ALARM_HISTORYFLUSH

Flushes on persistent storage all the history records still cached in memory buffers.

ALARM_HISTORYFLUSH ()

ALARM_STATSRESET

Resets all the alarms statistical information.

ALARM_STATSRESET ()

ALARM_STATSFLUSH

Flushes on persistent storage all the statistical information records still cached in memory buffers.

ALARM_STATSFLUSH ()

ALARM_EXPORT

Exports in a CSV file the records describing all the current active alarm instances.

NUMREC = **ALARM_EXPORT** (FILE)

input		
FILE :	ANY_STRING	name of the exported file

output		
NUMREC :	UDINT	number of exported records

ALARM_EXPORTHISTORY

Exports in a CSV file the records describing all the alarms currently logged in history.

NUMREC = **ALARM_EXPORTHISTORY** (FILE [, FROM [, TO]])

input			
FILE :	ANY_STRING		name of the exported file
FROM :	LDT	[OPTIONAL]	in case of limited time range, the initial timestamp of the needed interval; if given, only the records with times from this parameter onward will be included in the export
TO :	LDT	[OPTIONAL]	in case of limited time range, the final timestamp of the needed interval; if given, only the records with times up to this parameter will be included in the export

output		
NUMREC :	UDINT	number of exported records

ALARM_EXPORTSTATS

Exports in a CSV file the records containing the statistical information of all the project alarms.

NUMREC = **ALARM_EXPORTSTATS** (FILE)

input		
FILE :	ANY_STRING	name of the exported file

output		
NUMREC :	UDINT	number of exported records

ALARM_EXPORTCONFIG

Allows the customization of the list of fields included in exports.
Can affect active alarms, history and statistics exports.

ALARM_EXPORTCONFIG (KEYSACTIVE, KEYSHISTORY, KEYSSTATS)

input

KEYSACTIVE :	ANY_STRING	a string of keys, used to specify the list of fields (along with their order and format) for the active alarms exports
KEYSHISTORY :	ANY_STRING	a string of keys, used to specify the list of fields (along with their order and format) of the history records exports
KEYSSTATS :	ANY_STRING	a string of keys, used to specify the list of fields (along with their order and format) of alarms statistics exports

The function can be used to customize all three alarm contexts altogether, or even just one of them.
Simply giving an empty string in one of the parameters means that context must not be affected.

The supported fields and their corresponding keys are as follows.

KEYSACTIVE

name	N	name of the alarm
priority	P	priority of the alarm
group	G	name of the alarm's group
alarm type	A | a	type of the alarm; if the given key is an upper case '**A**', then the output is formatted as a numeric value; the possible values are: 1 : for instances of simple events, 2 : for instances of ISA alarms; if the given key is a lower case '**a**' instead, then the output is formatted as a readable string, that can be either: "Event" for instances of simple events, "ISA" for instances of ISA alarms
description	C	description message of the alarm instance; the produced string will be translated and will contain the values of the tags possibly included within the message configuration
quality	Q | q	quality flag of the alarm instance; the quality refers to the validity of the tags than might be embedded in the message; if the given key is an upper case '**Q**', then the output is formatted as a numeric value; the possible values are: 0 : the tags values are not valid, 1 : the tags values are valid; if the given key is a lower case '**q**' instead, then the output is formatted as a readable string, that can be either: "Ok" for valid tags values, "InvalidTags" for invalid tags values
on date	D	date of the alarm instance ON event
on time	T	time of the alarm instance ON event
on type	E | V	type of the alarm instance ON event;

		exists to match the set of information dedicated to the 2nd event (see below), but actually the only possible value for this field is “on”; if the given key is an ‘**E**’, then the output is formatted as a numeric value; the only possible value is: 0 : ON event; if the given key is a ‘**V**’ instead, then the output is formatted as a readable string, that can only be: “On”
on user	O	name of the user logged (usually but not necessarily on the server) when the alarm instance ON event was raised
on station	S	name of the machine (usually but not necessarily the server machine) that detected the alarm instance ON event
2nd date	d	date of the alarm instance secondary event
2nd time	t	time of the alarm instance secondary event
2nd type	e | v	type of the alarm instance secondary event; if existing, the information could be either “off” or “ack”; if the given key is an ‘**e**’, then the output is formatted as a numeric value; the possible values are: 1 : OFF event, 2 : ACK event; if the given key is a ‘**v**’ instead, then the output is formatted as a readable string, that can be either: “Off”, “Ack”
2nd user	o	name of the user logged when the alarm instance secondary event was detected (see above)
2nd station	s	name of the machine that detected the alarm instance secondary event (see above)

As a special case, it is possible to give a string that starts with a “@”, followed by few extra characters;
these characters can be given to change the behaviour and format of specific properties, only if already included in the currently configured keys (this is to be noted: the intent is to change behaviour of existing fields, not to add or remove fields; if the sequence of fields is to be changed, then regular keys strings are required).
The allowed extra characters are:
- “**A”** : the type of the alarm will become an upper case “A” and will be written as a numeric value;
- “**a**” : the type of the alarm will become a lower case “a” and will be written as a readable string;
- “**Q**” : the quality of the alarm tags will become an upper case “Q” and will be written as a numeric value;
- “**q**” : the quality of the alarm tags will become a lower case “q” and will be written as a readable string;
- “**E**” : the type of the ON event will become an “E” and will be written as a numeric value;
- “**V**” : the type of the ON event will become a “V” and will be written as a readable string;
- “**e**” : the type of the secondary event will become an “e” and will be written as a numeric value;
- “**v**” : the type of the secondary event will become a “v” and will be written as a readable string.

For example, if the current keys string is: “NPG**A**C**Q**DT**E**O”
and the following customization string is given: “@aqVv”
then the keys string becomes: “NPG**a**C**q**DT**V**O”
(where three fields have been changed, and one ignored because not part of the original keys).

KEYSHISTORY

name	N	name of the alarm
priority	P	priority of the alarm
group	G	name of the alarm’s group
alarm type	A | a	type of the alarm;

		if the given key is an upper case '**A**', then the output is formatted as a numeric value; the possible values are: 1 : for instances of simple events, 2 : for instances of ISA alarms; if the given key is a lower case '**a**' instead, then the output is formatted as a readable string, that can be either: "Event" for instances of simple events, "ISA" for instances of ISA alarms
description	C	description message of the alarm record; the produced string will be translated and will contain the values of the tags possibly included within the message configuration
quality	Q | q	quality flag of the alarm record; the quality refers to the validity of the tags than might be embedded in the message; if the given key is an upper case '**Q**', then the output is formatted as a numeric value; the possible values are: 0 : the tags values are not valid, 1 : the tags values are valid; if the given key is a lower case '**q**' instead, then the output is formatted as a readable string, that can be either: "Ok" for valid tags values, "InvalidTags" for invalid tags values
on date	D	date of the alarm event record
on time	T	time of the alarm event record
on type	E | e	type of the alarm event; if the given key is an '**E**', then the output is formatted as a numeric value; the possible values are: 0 : ON event; 1 : OFF event; 2 : ACK event; if the given key is an '**e**' instead, then the output is formatted as a readable string, that can be: "On" "Off" "Ack"
on user	O	name of the user logged (usually but not necessarily on the server) when the alarm instance ON event was raised
on station	S	name of the machine (usually but not necessarily the server machine) that detected the alarm instance ON event

As above, as a special case, it is possible to give a string that starts with a "@", followed by few extra characters;
these characters can be given to change the behaviour and format of specific fields, only if already included in the currently configured keys (this is to be noted: the intent is to change behaviour of existing fields, not to add or remove fields; if the sequence of fields is to be changed, then regular keys strings are required).
The allowed extra characters are:

- "**A**" : the type of the alarm will become an upper case "A" and will be written as a numeric value;
- "**a**" : the type of the alarm will become a lower case "a" and will be written as a readable string;
- "**Q**" : the quality of the alarm tags will become an upper case "Q" and will be written as a numeric value;
- "**q**" : the quality of the alarm tags will become a lower case "q" and will be written as a readable string;
- "**E**" : the type of the event will become an "E" and will be written as a numeric value;
- "**e**" : the type of the event will become an "e" and will be written as a readable string.

For example, if the current keys string is: "NPG**A**CDT**E**QOS"
and the following customization string is given: "@ae"
then the keys string becomes: "NPG**a**CDT**e**QOS".

KEYSSTATS

name	N	name of the alarm
priority	P	priority of the alarm
group	G	name of the alarm's group
description	C	description message of the alarm record; the produced string will be translated and will contain the values of the tags possibly included within the message configuration
state	E | e	current state of the alarm; if the given key is an upper case '**E**', then the output is formatted as a numeric value; the possible values are: 0 : OFF; 1 : ON; if the given key is a lower case '**e**' instead, then the output is formatted as a readable string, that can be: "Off" "On"
on number	B	number of times the alarm ON event occurred
on duration	R | r	how long the alarm remained in an ON state; if the given key is an upper case '**R**', then the output is formatted as a numeric value; the value is given in milliseconds; if the given key is a lower case '**r**' instead, then the output is formatted as a readable string, with the format "H:MM:SS.mmm" (hours:minutes:seconds.milliseconds)

As above, as a special case, it is possible to give a string that starts with a "@", followed by few extra characters;
these characters can be given to change the behaviour and format of specific fields, only if already included in the currently configured keys (this is to be noted: the intent is to change behaviour of existing fields, not to add or remove fields; if the sequence of fields is to be changed, then regular keys strings are required).
The allowed extra characters are:

- "**E**" : the state of the alarm will become an upper case "E" and will be written as a numeric value;
- "**e**" : the state of the alarm will become a lower case "e" and will be written as a readable string.
- "**R"** : the duration of the ON state will become an upper case "R" and will be written as a numeric value;
- "**r**" : the duration of the ON state will become a lower case "r" and will be written as a readable string.

For example, if the current keys string is: "NPG**e**B**r**"
and the following customization string is given: "@ER"
then the keys string becomes: "NPG**E**B**R**".

ALARM_PRINT

Prints the records describing all the current active alarm instances.

```
NUMREC = ALARM_PRINT ()
```

output		
NUMREC :	UDINT	number of printed records

ALARM_PRINTHISTORY

Prints the records describing all the alarms currently logged in history.

NUMREC = **ALARM_PRINTHISTORY** ()

output		
NUMREC :	UDINT	number of exported records

ALARM_PRINTSTATS

Prints the records containing the statistical information of all the project alarms.

NUMREC = **ALARM_PRINTSTATS** ()

output		
NUMREC :	UDINT	number of exported records

ALARM_GETNUMBER

Retrieves the total number of existing active alarm instances.

NUMBER = **ALARM_GETNUMBER** ([PRIORITY])

input

PRIORITY :	ANY_INT	[OPTIONAL]	minimum priority of alarms to be considered; if missing, there is no priority limit: all the alarms can be counted

output

NUMBER :	UDINT	number of alarms counted (number of active instances of alarms with at least the given priority)

ALARM_GETNUMISA

Retrieves the number of active instances of ISA alarms.

NUMBER = **ALARM_GETNUMISA** ([PRIORITY])

input

PRIORITY :	ANY_INT	[OPTIONAL]	minimum priority of alarms to be considered; if missing, there is no priority limit: all the alarms can be counted

output

NUMBER :	UDINT	number of alarms counted (number of active instances of ISA alarms with at least the given priority)

ALARM_GETNUMEVENTS

Retrieves the number of active instances of simple alarm events.

NUMBER = **ALARM_GETNUMEVENTS** ([PRIORITY])

input

PRIORITY :	ANY_INT	[OPTIONAL]	minimum priority of alarms to be considered; if missing, there is no priority limit: all the alarms can be counted

output

NUMBER :	UDINT	number of alarms counted (number of active instances of simple events with at least the given priority)

ALARM_GETNUMACK

Retrieves the number of active ISA alarm instances still waiting for an acknowledge.

NUMBER = **ALARM_GETNUMACK** ([PRIORITY])

input

PRIORITY :	ANY_INT	[OPTIONAL]	minimum priority of alarms to be considered; if missing, there is no priority limit: all the alarms can be counted

output

NUMBER :	UDINT	number of alarms counted (number of active ISA alarm instances waiting for acknowledge)

ALARM_GETNUMHISTORY

Retrieves the number of records logged in the alarms history.

```
NUMBER = ALARM_GETNUMHISTORY ()
```

output		
NUMBER :	UDINT	number of history records counted

ALARM_GETNUMINSTANCES

Retrieves the number of active instances existing for a given alarm.

NUMBER = **ALARM_GETNUMINSTANCES** (ALARM)

input		
ALARM :	ANY_STRING	name of the alarm

output		
NUMBER :	UDINT	number of counted instances

ALARM_GETINSTANCEID

Retrieves the ID of one of the active instances associated to a given alarm.

ID = **ALARM_GETINSTANCEID** (ALARM, INDEX)

input		
ALARM :	ANY_STRING	name of the alarm
INDEX :	ANY_INT	index (base-0) of the alarm instance; the index can range between 0 and **ALARM_GETNUMINSTANCES** (-1)

output		
ID :	UDINT	the ID of the active alarm instance

Together with the **ALARM_GETNUMINSTANCES**, this function is able to retrieve the IDs of all the instances currently existing for any given alarm.

ALARM_GETINFO

Retrieves a set of identification values for an alarm with a given name or ID.

ALARM_GETINFO ([NAME] | [ID])

input

NAME :	ANY_STRING	[OPTIONAL]	name of the alarm; can and must be given if and only if the ID is missing
ID :	ANY_INT	[OPTIONAL]	ID of the alarm; can and must be given if and only if the NAME is missing

The function must be called with a single parameter, either a string or a numeric one.
If the function is invoked with a string parameter, then the system assumes that the alarm is identified by its name; if the function is invoked with a numeric parameter, then the system assumes that the alarm is identified by its system ID.

The function doesn't directly return any information; a set of system variables is prepared instead, dedicated to a bunch of parameters related to the alarm configuration.
The interested variables are:

ALARM_NAME	the name of the alarm; might replicate the NAME parameter, if passed to the function;
ALARM_ID	the ID of the alarm; might replicate the ID parameter, if passed to the function;
ALARM_KEY	the custom key of the alarm;
ALARM_MESSAGE	the description message of the alarm, translated for the appropriate language; if the execution of the script was invoked by a client, then the language currently active in that client is used; if the execution of the script is instead triggered by a server event, then the server language is used; use **LANGUAGESET** and **LANGUAGEGET** to handle the language active in the server.

All the listed variables are assigned only if the function execution is successful; in case of errors (for example an invalid alarm name), then nothing is assigned or reset, and the variables will retain their old values.

example

```
_TRACE (ALARM_GETNAMEFROMKEY(11));
_TRACE (ALARM_GETMSGFROMKEY(11));
_TRACE (ANY_TO_STRING(ALARM_GETIDFROMKEY(11)));

ALARM_GETINFO (0);
_TRACE(ALARM_NAME);
_TRACE(ALARM_MESSAGE);
_TRACE(ANY_TO_STRING(ALARM_ID));
_TRACE(ANY_TO_STRING(ALARM_KEY));

ALARM_GETINFO ("Alarm2");
_TRACE(ALARM_NAME);
_TRACE(ALARM_MESSAGE);
_TRACE(ANY_TO_STRING(ALARM_ID));
_TRACE(ANY_TO_STRING(ALARM_KEY));
```

ALARM_GETNAMEFROMKEY

Retrieves the name of an alarm with a given custom key.

NAME = **ALARM_GETNAMEFROMKEY** (KEY)

input		
KEY :	ANY_INT	custom key of the alarm

output		
NAME :	WSTRING	system name of the alarm

ALARM_GETIDFROMKEY

Retrieves the ID of an alarm with a given custom key.

ID = **ALARM_GETIDFROMKEY** (KEY)

input		
KEY :	ANY_INT	custom key of the alarm

output		
ID :	UDINT	system ID of the alarm

ALARM_GETMSGFROMKEY

Retrieves the description message of an alarm with a given custom key.

MESSAGE = **ALARM_GETMSGFROMKEY** (KEY)

input		
KEY :	ANY_INT	custom key of the alarm

output		
MESSAGE :	WSTRING	description message of the alarm

The returned message string is translated for the appropriate language.
If the execution of the script was invoked by a client, then the language currently active in that client is used.
If the execution of the script is instead triggered by a server event, then the server language is used; use **LANGUAGESET** and **LANGUAGEGET** to handle the language active in the server.

ALARM_GETFIRSTPRJ

Retrieves information about the first alarm configured in the project, along with its events statistics.

ITER = **ALARM_GETFIRSTPRJ** ()

output		
ITER :	UDINT	an iterator to be used in subsequent calls to **ALARM_GETNEXTPRJ**; ≥ 1 if at least an alarm exists in the project (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

If a valid iterator is returned (≥ 1), it means at least an alarm exists, and information about the first one is stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the alarm
ALARM_RECKEY	(DINT)	: custom key of the alarm
ALARM_RECID	(UDINT)	: system ID of the alarm
ALARM_RECSTATE	(UDINT)	: state of the alarm (either 0 = OFF or 1 = ON)
ALARM_RECNUMINSTANCES	(UDINT)	: number of active instances currently registered for the alarm
ALARM_RECONNUMBER	(UDINT)	: number of active instances ever created for the alarm
ALARM_RECONDURATION	(LTIME)	: time ever spent by the system with this alarm in ON state

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no matching alarm found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

ALARM_GETNEXTPRJ

Retrieves information about the "next" alarm configured in the project, along with its events statistics.

ITER = **ALARM_GETNEXTPRJ** (LASTITER)

input

LASTITER :	UDINT	the iterator returned by the previous/last call to either an ALARM_GETFIRSTPRJ or ALARM_GETNEXTPRJ function (to be used to handle the progress of a whole acquisition loop)

output

ITER :	UDINT	an iterator to be used in further calls to ALARM_GETNEXTPRJ; ≥ 1 if "another" alarm has been found in the project (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

Can be used only after an acquisition loop has already been started with a call to ALARM_GETFIRSTPRJ.
Retrieves the alarm record following the one returned by the last call to an ALARM_GETFIRSTPRJ or ALARM_GETNEXTPRJ function.
Affects the exact same set of variables as the ALARM_GETFIRSTPRJ itself.
If a valid iterator is returned (≥ 1), it means a further alarm has been found in the project, and information about it has been stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the alarm
ALARM_RECKEY	(DINT)	: custom key of the alarm
ALARM_RECID	(UDINT)	: system ID of the alarm
ALARM_RECSTATE	(UDINT)	: state of the alarm (either 0 = OFF or 1 = ON)
ALARM_RECNUMINSTANCES	(UDINT)	: number of active instances currently registered for the alarm
ALARM_RECONNUMBER	(UDINT)	: number of active instances ever created for the alarm
ALARM_RECONDURATION	(LTIME)	: time ever spent by the system with this alarm in ON state

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no more matching alarm found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

example

```
VAR
   iter : UDINT;
END_VAR;

iter := ALARM_GETFIRSTPRJ ();
WHILE (iter > 0) DO
   IF ((ALARM_RECKEY >= 1000) AND (ALARM_RECKEY < 2000)) THEN  // look for all the alarms with a
      // ... do something ...                                  // custom key between 1000 and 1999
   END_IF;
   iter := ALARM_GETNEXTPRJ (iter);
END_WHILE;
```

ALARM_GETFIRSTON

Retrieves information about the first project alarm currently in an ON state.

ITER = **ALARM_GETFIRSTON** ()

output		
ITER :	UDINT	an iterator to be used in subsequent calls to **ALARM_GETNEXTON**; ≥ 1 if a (first) alarm in ON state has been found and returned (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

If a valid iterator is returned (≥ 1), it means at least an alarm in ON state exists, and information about the first one is stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the alarm
ALARM_RECKEY	(DINT)	: custom key of the alarm
ALARM_RECID	(UDINT)	: system ID of the alarm
ALARM_RECSTATE	(UDINT)	: state of the alarm (always 1 = ON)
ALARM_RECNUMINSTANCES	(UDINT)	: number of active instances currently registered for the alarm

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no matching alarm found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

→ Note: "**alarms in ON state**" are not to be confused with "**active alarms**":
whereas an "active alarm" is a registration of occurrence of an alarm that can either be still ON or still to be acknowledged by a user (note that for each project alarm, multiple active registrations could exist at any given time), an "alarm in ON state" is simply a project alarm whose activation condition is still present (these alarms have at least one active registration, and their last registration has not received an OFF notification yet).
The difference applies to ISA alarms only; an example:
- alarm_1 ON event → the alarm is now ON; 1 active registration is created
- alarm_1 OFF event → the alarm is now OFF; 1 active registration still exists (not removed by an acknowledge)
- alarm_1 ON event → the alarm is now ON; 2 active registrations now exist (the ON event creates a new one)
- alarm_1 OFF event → the alarm is now OFF; 2 active registrations still exist
- a global acknowledge occurs → the active registrations are removed

ALARM_GETNEXTON

Retrieves information about the "next" project alarm currently in an ON state.

ITER = **ALARM_GETNEXTON** (LASTITER)

input		
LASTITER :	UDINT	the iterator returned by the previous/last call to either an ALARM_GETFIRSTON or ALARM_GETNEXTON function (to be used to handle the progress of a whole acquisition loop)

output		
ITER :	UDINT	an iterator to be used in further calls to ALARM_GETNEXTON; ≥ 1 if "another" alarm in ON state has been found and returned (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

Can be used only after an acquisition loop has already been started with a call to ALARM_GETFIRSTON.
Retrieves the matching alarm record following the one returned by the last call to an ALARM_GETFIRSTON or ALARM_GETNEXTON function.
Affects the exact same set of variables as the ALARM_GETFIRSTON itself.
If a valid iterator is returned (≥ 1), it means a further alarm in ON state has been found, and information about it has been stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the alarm
ALARM_RECKEY	(DINT)	: custom key of the alarm
ALARM_RECID	(UDINT)	: system ID of the alarm
ALARM_RECSTATE	(UDINT)	: state of the alarm (always 1 = ON)
ALARM_RECNUMINSTANCES	(UDINT)	: number of active instances currently registered for the alarm

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no more matching alarm found).
See the <VARIABLES> section for the complete list of the ALARM_REC$$$ variables.

See the ALARM_GETNEXTPRJ description for an *example* about the usage of these ALARM_GETFIRST$$$ and ALARM_GETNEXT$$$ pairs in loop.

ALARM_GETFIRSTACTIVE

Retrieves information about the first active alarm registration currently stored.

ITER = **ALARM_GETFIRSTACTIVE** ()

output		
ITER :	UDINT	an iterator to be used in subsequent calls to ALARM_GETNEXTACTIVE; ≥ 1 if a (first) active alarm registration has been found and returned (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

If a valid iterator is returned (≥ 1), it means at least an active alarm registration exists, and information about the first one is stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECINSTANCEID	(UDINT)	: system ID of the active alarm registration
ALARM_RECSTATE	(UDINT)	: state of the registration (0 = OFF, 1 = ON, 2 = ON+OFF, 3 = ON+ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the registration
ALARM_RECONTIME	(LDT)	: timestamp of the registration ON event
ALARM_RECONUSER	(WSTRING)	: username recorded with the registration ON event
ALARM_RECALTTIME	(LDT)	: timestamp of the registration secondary (OFF/ACK) event
ALARM_RECALTUSER	(WSTRING)	: username recorded with the registration secondary (OFF/ACK) event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no matching alarm found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

→ Note: "**active alarms**" are not to be confused with "**alarms in ON state**":
whereas an "active alarm" is a registration of occurrence of an alarm that can either be still ON or still to be acknowledged by a user (note that for each project alarm, multiple active registrations could exist at any given time), an "alarm in ON state" is simply a project alarm whose activation condition is still present (these alarms have at least one active registration, and their last registration has not received an OFF notification yet).
The difference applies to ISA alarms only; an example:
- alarm_1 ON event → the alarm is now ON; 1 active registration is created
- alarm_1 OFF event → the alarm is now OFF; 1 active registration still exists (not removed by an acknowledge)
- alarm_1 ON event → the alarm is now ON; 2 active registrations now exist (the ON event creates a new one)
- alarm_1 OFF event → the alarm is now OFF; 2 active registrations still exist
- a global acknowledge occurs → the active registrations are removed

ALARM_GETNEXTACTIVE

Retrieves information about the "next" active alarm registration currently stored.

ITER = **ALARM_GETNEXTACTIVE** (LASTITER)

input		
LASTITER :	UDINT	the iterator returned by the previous/last call to either an ALARM_GETFIRSTACTIVE or ALARM_GETNEXTACTIVE function (to be used to handle the progress of a whole acquisition loop)

output		
ITER :	UDINT	an iterator to be used in further calls to ALARM_GETNEXTACTIVE; ≥ 1 if "another" active alarm registration has been found and returned (0 otherwise); the information about the retrieved alarm is stored in a set of predefined variables; see details below

Can be used only after an acquisition loop has already been started with a call to ALARM_GETFIRSTACTIVE.
Retrieves the matching alarm record following the one returned by the last call to an ALARM_GETFIRSTACTIVE or ALARM_GETNEXTACTIVE function.
Affects the exact same set of variables as the ALARM_GETFIRSTACTIVE itself.
If a valid iterator is returned (≥ 1), it means a further active alarm registration has been found, and information about it has been stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECINSTANCEID	(UDINT)	: system ID of the active alarm registration
ALARM_RECSTATE	(UDINT)	: state of the registration (0 = OFF, 1 = ON, 2 = ON+OFF, 3 = ON+ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the registration
ALARM_RECONTIME	(LDT)	: timestamp of the registration ON event
ALARM_RECONUSER	(WSTRING)	: username recorded with the registration ON event
ALARM_RECALTTIME	(LDT)	: timestamp of the registration secondary (OFF/ACK) event
ALARM_RECALTUSER	(WSTRING)	: username recorded with the registration secondary (OFF/ACK) event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no more matching alarm found).
See the <VARIABLES> section for the complete list of the ALARM_REC$$$ variables.

See the ALARM_GETNEXTPRJ description for an *example* about the usage of these ALARM_GETFIRST$$$ and ALARM_GETNEXT$$$ pairs in loop.

ALARM_GETFIRSTHISTORY

Retrieves information about the first alarm history record currently stored.

ITER = **ALARM_GETFIRSTHISTORY** ()

output

ITER :	UDINT	an iterator to be used in subsequent calls to **ALARM_GETNEXTHISTORY**; ≥ 1 if a (first) alarm history record has been found and returned (0 otherwise); the information about the retrieved record is stored in a set of predefined variables; see details below

If a valid iterator is returned (≥ 1), it means at least an alarm history record exists, and information about the first one is stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECEVENTTYPE	(UDINT)	: type of the recorded alarm event (0 = ON, 1 = OFF, 2 = ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the alarm
ALARM_RECEVENTTIME	(LDT)	: timestamp of the alarm event
ALARM_RECEVENTUSER	(WSTRING)	: username recorded with the alarm event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no matching record found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

ALARM_GETNEXTHISTORY

Retrieves information about the "next" alarm history record currently stored.

ITER = **ALARM_GETNEXTHISTORY** (LASTITER)

input

LASTITER :	UDINT	the iterator returned by the previous/last call to either an ALARM_GETFIRSTHISTORY or ALARM_GETNEXTHISTORY function (to be used to handle the progress of a whole acquisition loop)

output

ITER :	UDINT	an iterator to be used in further calls to ALARM_GETNEXTHISTORY; ≥ 1 if "another" alarm history record has been found and returned (0 otherwise); the information about the retrieved record is stored in a set of predefined variables; see details below

Can be used only after an acquisition loop has already been started with a call to ALARM_GETFIRSTHISTORY.
Retrieves the matching alarm record following the one returned by the last call to an ALARM_GETFIRSTHISTORY or ALARM_GETNEXTHISTORY function.
Affects the exact same set of variables as the ALARM_GETFIRSTHISTORY itself.
If a valid iterator is returned (≥ 1), it means a further alarm history record has been found, and information about it has been stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECEVENTTYPE	(UDINT)	: type of the recorded alarm event (0 = ON, 1 = OFF, 2 = ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the alarm
ALARM_RECEVENTTIME	(LDT)	: timestamp of the alarm event
ALARM_RECEVENTUSER	(WSTRING)	: username recorded with the alarm event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no more matching record found).
See the <VARIABLES> section for the complete list of the ALARM_REC$$$ variables.

See the ALARM_GETNEXTPRJ description for an *example* about the usage of these ALARM_GETFIRST$$$ and ALARM_GETNEXT$$$ pairs in loop.

ALARM_GETFIRSTHPACK

Retrieves information about the first alarm packed history record currently stored.

ITER = **ALARM_GETFIRSTHPACK** ([NEWFIRST])

input

NEWFIRST :	BOOL	[OPTIONAL]	TRUE if the browse loop is meant to return records sorted from the newest to the oldest; if missing the records start from the oldest

output

ITER :	UDINT	an iterator to be used in subsequent calls to **ALARM_GETNEXTHPACK**; ≥ 1 if a (first) record has been found and returned (0 otherwise); the information about the retrieved record is stored in a set of predefined variables; see details below

"Packed history records" refer to alarm records built using matching couples of basic history records; each "packed record" contains information about matching ON event and OFF event of the same alarm instance.
Most of the information come from the ON event record, while the OFF event is used to extend the information with the timestamp of the OFF event itself.
The functionality is mainly meant for simple events management, or at least for ISA alarms with "single instance" behaviour enabled: ACK events (if present) are ignored, and multi-instance alarms might cause inconsistencies among the retrieved data.

If a valid iterator is returned (≥ 1), it means at least a "alarm packed history record" exists, and information about the first one is stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECEVENTTYPE	(UDINT)	: type of the recorded alarm event (0 = ON, 1 = OFF, 2 = ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the alarm
ALARM_RECEVENTUSER	(WSTRING)	: username recorded with the alarm event
ALARM_RECONTIME	(LDT)	: timestamp of the ON event
ALARM_RECALTTIME	(LDT)	: timestamp of the OFF event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no matching record found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

ALARM_GETNEXTHPACK

Retrieves information about the "next" alarm packed history record currently stored.

ITER = **ALARM_GETNEXTHPACK** (LASTITER [, NEWFIRST])

input

LASTITER :	UDINT	the iterator returned by the previous/last call to either an ALARM_GETFIRSTHPACK or ALARM_GETNEXTHPACK function (to be used to handle the progress of a whole acquisition loop)
NEWFIRST :	BOOL	[OPTIONAL] TRUE if the browse loop is meant to return records sorted from the newest to the oldest if missing the records start from the oldest

output

ITER :	UDINT	an iterator to be used in further calls to ALARM_GETNEXTHPACK; ≥ 1 if "another" record has been found and returned (0 otherwise); the information about the retrieved record is stored in a set of predefined variables; see details below

See ALARM_GETFIRSTHPACK for information about “packed history records”.

Can be used only after an acquisition loop has already been started with a call to ALARM_GETFIRSTHPACK.
Retrieves the matching alarm record following the one returned by the last call to an ALARM_GETFIRSTHPACK or ALARM_GETNEXTHPACK function.
Affects the exact same set of variables as the ALARM_GETFIRSTHPACK itself.
If a valid iterator is returned (≥ 1), it means a further “alarm packed history record” has been found, and information about it has been stored in the following set of predefined variables:

ALARM_RECNAME	(WSTRING)	: name of the associated alarm
ALARM_RECKEY	(DINT)	: custom key of the associated alarm
ALARM_RECID	(UDINT)	: system ID of the associated alarm
ALARM_RECEVENTTYPE	(UDINT)	: type of the recorded alarm event (0 = ON, 1 = OFF, 2 = ACK)
ALARM_RECMESSAGE	(WSTRING)	: description message of the alarm
ALARM_RECEVENTUSER	(WSTRING)	: username recorded with the alarm event
ALARM_RECONTIME	(LDT)	: timestamp of the ON event
ALARM_RECALTTIME	(LDT)	: timestamp of the OFF event

No other predefined system variable is affected by the function;
no predefined variable at all is affected if the function returns 0 (no more matching record found).
See the <VARIABLES> section for the complete list of the **ALARM_REC$$$** variables.

See the ALARM_GETNEXTPRJ description for an *example* about the usage of these ALARM_GETFIRST$$$ and ALARM_GETNEXT$$$ pairs in loop.

< VARIABLES >

The following are the variables usable to share information and directives for the alarms management:

Variables for identifiers info

ALARM_NAME
type WSTRING
access R
gives the name of the alarm treated by the last successful execution of the function **ALARM_GETINFO**

ALARM_ID
type UDINT
access R
gives the ID of the alarm treated by the last successful execution of the function **ALARM_GETINFO**

ALARM_KEY
type DINT
access R
gives the custom key of the alarm treated by the last successful execution of the function **ALARM_GETINFO**

ALARM_MESSAGE
type WSTRING
access R
gives the description message of the alarm treated by the last successful execution of the function **ALARM_GETINFO**;
the message is translated with the most appropriate language (from either a client or the server), depending on the state of the caller; see the mentioned function for details

Variables for browsed records

- all the following variables are prepared by successful executions of any function of the family *ALARM_GETFIRST/NEXTxxx* (ALARM_GETFIRSTPRJ, ALARM_GETNEXTPRJ, ALARM_GETFIRSTON, ALARM_GETNEXTON, ALARM_GETFIRSTACTIVE, ALARM_GETNEXTACTIVE, ALARM_GETFIRSTHISTORY, ALARM_GETNEXTHISTORY); failed executions won't affect them;
- not every function will affect them all: depending on the carried information, some are reserved to specific query contexts (see details in the table below);
- message-related variables are translated with the most appropriate language (from either a client or the server), depending on the state of the caller;
- record "states" can have different set of values depending on the involved alarms context;
- more detailed information can be found in the description of each browsing function, and among the notes after the table below.

variable name	type	contexts
ALARM_RECNAME	WSTRING	-PRJ , -ON , -ACTIVE , -HISTORY , -HPACK
ALARM_RECKEY	DINT	-PRJ , -ON , -ACTIVE , -HISTORY , -HPACK
ALARM_RECID	UDINT	-PRJ , -ON , -ACTIVE , -HISTORY , -HPACK
ALARM_RECSTATE	UDINT	-PRJ , -ON , -ACTIVE
ALARM_RECNUMINSTANCES	UDINT	-PRJ , -ON
ALARM_RECONNUMBER	UDINT	-PRJ
ALARM_RECONDURATION	LTIME	-PRJ
ALARM_RECINSTANCEID	UDINT	-ACTIVE
ALARM_RECMESSAGE	WSTRING	-ACTIVE , -HISTORY , -HPACK
ALARM_RECONTIME	LDT	-ACTIVE , -HPACK
ALARM_RECONUSER	WSTRING	-ACTIVE
ALARM_RECALTTIME	LDT	-ACTIVE , -HPACK
ALARM_RECALTUSER	WSTRING	-ACTIVE
ALARM_RECEVENTTYPE	UDINT	-HISTORY , -HPACK
ALARM_RECEVENTTIME	LDT	-HISTORY
ALARM_RECEVENTUSER	WSTRING	-HISTORY , -HPACK

ALARM_RECNAME

type WSTRING
access R
affected by browsing functions for the contexts: -PRJ, -ON, -ACTIVE, -HISTORY
gives the name of the alarm associated to the retrieved record

ALARM_RECKEY

type DINT
access R
affected by browsing functions for the contexts: -PRJ, -ON, -ACTIVE, -HISTORY
gives the custom key identifier of the alarm associated to the retrieved record

ALARM_RECID

type UDINT
access R
affected by browsing functions for the contexts: -PRJ, -ON, -ACTIVE, -HISTORY
gives the system ID of the alarm associated to the retrieved record

ALARM_RECSTATE

type UDINT
access R
affected by browsing functions for the contexts: -PRJ, -ON, -ACTIVE
gives the state of the retrieved record; could have different meaning depending on the specific alarms context:
for -PRJ: 0 = **OFF**, 1 = **ON**
for -ON: 1 = **ON**
for -ACTIVE: 0 = **OFF**, 1 = **ON**, 2 = **ON+OFF**, 3 = **ON+ACK**

ALARM_RECNUMINSTANCES
type UDINT
access R
affected by browsing functions for the contexts: -**PRJ**, -**ON**
gives the number of active instances currently recorded for the referenced alarm

ALARM_RECONNUMBER
type UDINT
access R
affected by browsing functions for the contexts: -**PRJ**
gives the number of ON events counted for the referenced alarm since the start of the statistics collection

ALARM_RECONDURATION
type LTIME
access R
affected by browsing functions for the contexts: -**PRJ**
gives the total time the referenced alarm has spent in ON state since the start of the statistics collection

ALARM_RECINSTANCEID
type UDINT
access R
affected by browsing functions for the contexts: -**ACTIVE**
gives the instance ID of the retrieved active alarm registration

ALARM_RECMESSAGE
type WSTRING
access R
affected by browsing functions for the contexts: -**ACTIVE**, -**HISTORY**
gives the alarm message built for the retrieved record, translated using the most appropriate language (from either a client or the server), depending on the state of the caller

ALARM_RECONTIME
type LDT
access R
affected by browsing functions for the contexts: -**ACTIVE**
gives the timestamp of the ON event of the retrieved active instance

ALARM_RECONUSER
type WSTRING
access R
affected by browsing functions for the contexts: -**ACTIVE**
gives the username recorded with the ON event of the retrieved active instance

ALARM_RECALTTIME
type LDT
access R
affected by browsing functions for the contexts: -**ACTIVE**
gives the timestamp of the secondary event (either an OFF or an ACK event) of the retrieved active instance; returns 0 if only the ON event exists

ALARM_RECALTUSER
type WSTRING
access R
affected by browsing functions for the contexts: -**ACTIVE**
gives the username recorded with the secondary event (either an OFF or an ACK event) of the retrieved active instance

ALARM_RECEVENTTYPE
type UDINT
access R
affected by browsing functions for the contexts: -**HISTORY**
gives the event identifier of the retrieved history record;

expected values are: 0 = **ON**, 1 = **OFF**, 2 = **ACK**

ALARM_RECEVENTTIME

type LDT
access R
affected by browsing functions for the contexts: -**HISTORY**
gives the timestamp of the retrieved history record

ALARM_RECEVENTUSER

type WSTRING
access R
affected by browsing functions for the contexts: -**HISTORY**
gives the username stored with the retrieved history record

14. RUNTIME - RECIPES

RECIPE_LOAD

Loads a given recipe from archive to buffer.

RECIPE_LOAD (STRUCTURE, RECIPE [, NOLOG])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING		the name of the needed recipe
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

RECIPE_SAVE

Saves in archive the recipe currently in buffer.

RECIPE_SAVE (STRUCTURE [, RECIPE [, NOLOG]])

input

STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING	[OPTIONAL]	a new name for the saved recipe if missing (or empty), the recipe is saved with the name currently in buffer
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

The optional parameter allows the method to work both as a "Save" and as a "SaveAs" instruction:
giving an explicit name for the recipe will force the new name to be stored in the name buffer tag before the recipe is saved; otherwise the name already in the buffer is used.
This also means that if a name is not given, then in the buffer there MUST be a valid name, otherwise the call fails.

In both cases (with or without a recipe name explicitly given) existing recipes will be silently overwritten.
In case overwrites have to be avoided, it is responsibility of the programmer to check the recipes validity with information from **RECIPE_EXIST** and **RECIPE_GETCURNAME**.

example

```
VAR
   rcstr : WSTRING [64];
   rcnum : UDINT;
END_VAR;

rcstr := 'Recipe1';

rcnum := RECIPE_GETNUMBER (rcstr);     // = 0 (starting with empty archive)
RECIPE_SAVE (rcstr, 'AA');             // save new recipe
rcnum := RECIPE_GETNUMBER (rcstr);     // = 1 (AA added)
RECIPE_SAVE (rcstr, 'BB');             // save new recipe
rcnum := RECIPE_GETNUMBER (rcstr);     // = 2 (BB added)
RECIPE_SAVE (rcstr);                   // re-save current recipe (BB)
rcnum := RECIPE_GETNUMBER (rcstr);     // = 2 (BB overwritten)
RECIPE_SAVE (rcstr, 'AA');             // re-save AA
rcnum := RECIPE_GETNUMBER (rcstr);     // = 2 (AA overwritten)
```

RECIPE_DOWNLOAD

Transfers a recipe from archive to device.

RECIPE_DOWNLOAD (STRUCTURE, RECIPE [, SYNC [, NOLOG]])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING		the name of the involved recipe
SYNC :	BOOL	[OPTIONAL]	TRUE if the transfer must be synchronized; FALSE otherwise; if missing, the default is FALSE (no synchronization)
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

The device tags are written with values taken directly from the archive, without affecting the buffer tags.

Note that the download operation involves communication with external devices and could take up a long time to complete.
Transfers are blocking for the calling script: this function terminates only after the whole transfer has been completed, so the programmer must be aware that the script will be standing in wait for its whole duration.
See **RECIPE_TRANSFERBUSY** and **RECIPE_TRANSFERWAIT** for information about transfers synchronization issues.

RECIPE_UPLOAD

Transfers a recipe from device to archive.

RECIPE_UPLOAD (STRUCTURE [, RECIPE [, SYNC [, NOLOG]]])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING	[OPTIONAL]	the name used to save the recipe in the archive; can be used to force a specific name for the stored recipe, and is mandatory if there is no other name source for the uploaded recipe; can be omitted instead (or given empty) if the recipe in the device already has a name (if the recipe structure already includes a name tag on device);
SYNC :	BOOL	[OPTIONAL]	TRUE if the transfer must be synchronized; FALSE otherwise; if missing, the default is FALSE (no synchronization)
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

The values of the device tags are read and stored directly in the archive, without affecting the buffer tags.

Note that in order for this method to be usable, a recipe name must be somehow available;
the possible name sources are:
- a name tag pre-configured by the recipe structure on the device (the recipe needs to include its name mapped on a device tag): in this case the name will be read from the device when the recipe is uploaded;
- the optional RECIPE parameter passed to this function.

If the optional name is passed here, then it has precedence over the device name tag.
If there is no RECIPE parameter instead, or if an empty string is given, then the name in the device tag is used.
If none of the two is available, then an error is issued.

Whatever the source, the name used is assigned to the recipe and stored along with it.
Existing recipes (already in archive with matching name) will be silently overwritten.

Note that the upload operation involves communication with external devices and could take up a long time to complete.
Transfers are blocking for the calling script: this function terminates only after the whole transfer has been completed, so the programmer must be aware that the script will be standing in wait for its whole duration.
See **RECIPE_TRANSFERBUSY** and **RECIPE_TRANSFERWAIT** for information about transfers synchronization issues.

RECIPE_DOWNLOADBUF

Transfers a recipe from buffer to device.

RECIPE_DOWNLOADBUF (STRUCTURE [, SYNC [, NOLOG]])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
SYNC :	BOOL	[OPTIONAL]	TRUE if the transfer must be synchronized; FALSE otherwise; if missing, the default is FALSE (no synchronization)
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

The current values of the buffer tags are written in the device; the archive is not affected by the operation.

Note that the download operation involves communication with external devices and could take up a long time to complete.
Transfers are blocking for the calling script: this function terminates only after the whole transfer has been completed, so the programmer must be aware that the script will be standing in wait for its whole duration.
See **RECIPE_TRANSFERBUSY** and **RECIPE_TRANSFERWAIT** for information about transfers synchronization issues.

RECIPE_UPLOADBUF

Transfers a recipe from device to buffer.

RECIPE_UPLOADBUF (STRUCTURE [, SYNC [, NOLOG]])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
SYNC :	BOOL	[OPTIONAL]	TRUE if the transfer must be synchronized; FALSE otherwise; if missing, the default is FALSE (no synchronization)
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

The values of the device tags are read and copied in the buffer tags; the archive is not affected by the operation.

Note that the download operation involves communication with external devices and could take up a long time to complete.
Transfers are blocking for the calling script: this function terminates only after the whole transfer has been completed, so the programmer must be aware that the script will be standing in wait for its whole duration.
See **RECIPE_TRANSFERBUSY** and **RECIPE_TRANSFERWAIT** for information about transfers synchronization issues.

RECIPE_TRANSFERBUSY

Checks whether a recipe transfer is currently in progress.

BUSY = **RECIPE_TRANSFERBUSY** ()

output

BUSY :	BOOL	TRUE if a transfer is in progress; FALSE otherwise

In other words, this function states whether the recipes transfers engine is busy or not; checks are in order since only one (asynchronous) transfer at a time can be executed.
This function is meant to be used along with the transfer functions (**RECIPE_UPLOAD**, **RECIPE_DOWNLOAD**, **RECIPE_UPLOADBUF**, **RECIPE_DOWNLOADBUF**) to allow the synchronization of scripts and predefined functions, or simply to let the programmer to implement some way to show information about the panel activity.

About the transfers synchronizations

All the transfer functions called by scripts are blocking, meaning they guarantee that their operation is complete when they exit to pass to the next script step. So, from this point of view, all the scripts operations are handled synchronously.
Transfers, though, can be requested and executed by different contexts, not directly connected to the scripts: they could be requested by events predefined functions, by recipe views menu commands, and so on; so it is actually possible to reach a point in a script where a transfer request has to be submitted, while another is already in progress.
This **RECIPE_TRANSFERBUSY** allows the script to be aware of this situation, and handle the busy state accordingly.

example

```
FUNCTION WaitForTransfer
   WHILE RECIPE_TRANSFERBUSY() DO
      SLEEP (1);
   END_WHILE;
END_FUNCTION;

WaitForTransfer ();
RECIPE_DOWNLOAD ('Recipe1', FALSE);
```

See also **RECIPE_TRANSFERWAIT** for another tool to handle transfers' asynchronicity.

example

```
RECIPE_TRANSFERWAIT ();
RECIPE_UPLOAD ('Recipe1', FALSE);
```

RECIPE_TRANSFERWAIT

Waits for the termination of a transfer currently in progress.

WAITED = **RECIPE_TRANSFERWAIT** ()

output

WAITED :	BOOL	TRUE if the function actually waited for a transfer in progress; FALSE if there was nothing to wait for

When called, this function blocks the script until the transfer in progress has been completed; if there is no transfer in progress, the function immediately exits.

Like the **RECIPE_TRANSFERBUSY**, this function is used to handle the synchronization of recipes transfers and other script operations (needed when transfers could have been executed by different contexts); see **RECIPE_TRANSFERBUSY** information and issues on the matter.

RECIPE_DELETE

Deletes a recipe (or all the recipes) from a structure archive.

RECIPE_DELETE (STRUCTURE [, RECIPE [, NOLOG]])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING	[OPTIONAL]	name of the recipe that has to be removed if missing (or empty), then ALL the recipes of the structure are removed
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

about deletions

The recipes structures archives are organized in "records": every time a new recipe is stored, a record is taken up; every time a recipe is deleted a used record is released.
Note though that "released" records are not removed from the archive: they remain as "reusable" slots; they still take up space in the archive file, but they can be used in the future by new saved recipes.
This mechanic allows better performances in the recipes load/save operations, and allows all the existing recipes to retain their IDs (defined as base-1 indexes of their position in the archive).

For example:
- if recipes A and B are saved in an empty archive, they will take its 1st and 2nd records;
- if now recipe A is deleted, then the 1st record becomes free, and the recipe B is still in the 2nd record;
- if now a new recipe C is saved, it is stored in the 1st record, recognized as free.

Programmers must be aware of this mechanics, since some of the scripting instructions make use of it.
For example, the function **RECIPE_GETNUMBER** is used to retrieve the number of valid recipes stored in an archive, while the function **RECIPE_GETRECORDS** is used to retrieve the total number or records, that includes records used by valid recipes and records released and retained as free.
Also, as another example, the function **RECIPE_GETINFO** makes use of the mentioned records IDs (indexes): the programmer must know how the given IDs could point to either actual recipes or free archive slots.

Remember that it is always possible to explicitly request a cleanup of the archives records: the instruction **RECIPE_PACKARCHIVE** removes from the archive all the empty slots and reorganize the valid recipes records in a continuous sequence.
For example:
- if recipes A and B are stored in the 1st and 3rd records, while the 2nd record is currently free,
- after a **RECIPE_PACKARCHIVE** directive, the free slot disappears, and the recipes simply take the 1st and 2nd record.
The programmer must be aware of the fact that after a pack request, the IDs of the recipes will be different.

A final note: to allow the best possible performances of the archive accesses, after a **RECIPE_DELETE** is called to remove all the recipes of a structure (without the 2nd parameter), a further call to **RECIPE_PACKARCHIVE** is always recommended.

Several *examples* throughout this document are making use of this function; see especially **RECIPE_GETNUMBER** and **RECIPE_PACKARCHIVE**.

RECIPE_RENAME

Renames a recipe existing in a structure archive.

RECIPE_RENAME (STRUCTURE, OLDNAME, NEWNAME [, NOLOG])

input

STRUCTURE :	ANY_STRING		the name of the involved structure
OLDNAME :	ANY_STRING		the old name of the recipe
NEWNAME :	ANY_STRING		the new name of the recipe
NOLOG :	BOOL	[OPTIONAL]	a flag usable to inhibit the logging of the associated runtime event by the FDA auditor; TRUE means the event will not be logged; if missing, the default is FALSE (event logged)

RECIPE_PACKARCHIVE

Compacts the records of a structure archive.

RECIPE_PACKARCHIVE (STRUCTURE)

input

STRUCTURE :	ANY_STRING	the name of the involved structure

This method is used to remove all the free records from an archive and compact the remaining valid recipes in a continuous sequence of records with consecutive IDs.

For example:
- start from an empty archive and add recipes A, B and C (RECIPE_SAVE)
  the archive records will be: **1** (A) , **2** (B) , **3** (C)
  the retrievable records information will be: RECIPE_GETNUMBER() = 3 , RECIPE_GETRECORDS() = 3
- now remove the recipe B (RECIPE_DELETE)
  the archive records will be: **1** (A) , 2 (free) , **3** (C)
  the retrievable records information will be: RECIPE_GETNUMBER() = 2 , RECIPE_GETRECORDS() = 3
- now execute an archive pack (RECIPE_PACKARCHIVE)
  the archive records will be: **1** (A) , **2** (C)
  the retrievable records information will be: RECIPE_GETNUMBER() = 2 , RECIPE_GETRECORDS() = 2

As seen in this last step, the programmer must be aware of the fact that the RECIPE_PACKARCHIVE might change the IDs of the existing recipes (again: defined as base-1 indexes of their records within the archive).

See RECIPE_DELETE for notes regarding the management of deleted recipes and released records.

example

```
VAR
   rcstr : WSTRING [64] := 'Recipe1';
   rnrec, rntot : UDINT;
END_VAR;

rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 0,0 (empty archive)
RECIPE_SAVE (rcstr, 'AA');                                               // adding recipe AA
RECIPE_SAVE (rcstr, 'BB');                                               // adding recipe BB
RECIPE_SAVE (rcstr, 'CC');                                               // adding recipe CC
rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 3,3 (3 recipes available)
RECIPE_DELETE (rcstr, 'BB');                                             // deleting recipe BB
rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 2,3 (2 recipes + 1 free)
rcnum := RECIPE_GETID (rcstr, 'CC');                                     // = 3 (still in 3rd record)
RECIPE_PACKARCHIVE (rcstr);                                              // packing the archive
rcnum := RECIPE_GETID (rcstr, 'CC');                                     // = 2 (now in 2nd record)
rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 2,2 (no more free slots)
RECIPE_DELETE (rcstr);                                                   // deleting all the recipes
rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 0,2 (2 reusable records)
RECIPE_PACKARCHIVE (rcstr);                                              // packing again the archive
rnrec := RECIPE_GETNUMBER (rcstr);  rntot := RECIPE_GETRECORDS (rcstr);  // = 0,0 (everything cleaned up)
```

RECIPE_CLEARBUFFER

Resets the values of all the buffer tags of a given structure.

RECIPE_CLEARBUFFER (STRUCTURE)

input

STRUCTURE :	ANY_STRING	the name of the involved structure

RECIPE_COMPARE

Compares the content of two recipes of a given structure, to see whether they are identical or not.

DIFFERENT = **RECIPE_COMPARE** (STRUCTURE, RECIPE1, RECIPE2)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE1 :	ANY_STRING	the name of one of the recipes to be compared
RECIPE2 :	ANY_STRING	the name of the other recipe to be compared

output		
DIFFERENT :	BOOL	TRUE if the compared recipes are different; FALSE if they are identical

The comparison only takes into account the user-defined fields values: standard information stored within the recipe, such as recipe name, comment, ID and timestamp are ignored of course.

RECIPE_COMPARESET

Compares the content of a recipe in archive with the recipe values currently stored in a device or buffer tags set.

DIFFERENCE = **RECIPE_COMPARESET** (STRUCTURE, RECIPE, USEDEVICE, RELOADDEVICE [, RESULTTAG])

input

STRUCTURE :	ANY_STRING		the name of the involved structure
RECIPE :	ANY_STRING		the name of the recipe already in archive
USEDEVICE :	BOOL		FALSE if the recipe in archive has to be compared with the current buffer tags set; TRUE if it has to be compared with the device tags set
RELOADDEVICE :	BOOL		only meaningful if the selected tags set is the device (if USEDEVICE is TRUE); ignored otherwise; TRUE is used to state that the current device tags have to be updated with a new recipe upload before the comparison is executed (ensure the comparison is done with updated values from the device); FALSE means the comparison has to be done with the values already in the device tags
RESULTTAG :	ANY_STRING	[OPTIONAL]	the name of the tag that has to be used to store precise comparison information about each recipe field; if existing, the tag must be an array of boolean elements, ideally with enough elements to hold information about the recipe name, comment, and all its configured fields; if explicitly given as an empty string, the system automatically stores the result in the implicit comparison tag-array automatically created by Crew™; if omitted, the result is not stored in a tag, but simply returned by the function

output

DIFFERENCE :	ANY	the returned value is actually an ARRAY [] OF BOOL, but its dimension is not fixed since it will depend on the number of fields in the indicated recipe; the precise number of fields should be equal to the number of recipe fields + 3 (3 more elements are reserved at the beginning of the array, for difference information about the recipe overall, the recipe name and the recipe comment); if an output tag name is given though, the number of elements of the array will match that of the tag itself

The recipe already in archive isn't transferred anywhere (buffer tags are not affected by the comparison).

As already explained, it is possible to execute the comparison between a recipe in archive and:
- the recipe currently stored in the buffer tags,
- the recipe currently stored in the device tags,
- the recipe currently present on the device, preemptively uploaded in the device tags.

The result tag is optional; if given, it must be an array of boolean elements (ideally big enough to hold difference information about all the recipe fields). In this tag is automatically written an exact copy of the DIFFERENCE value returned by the function.

The value returned by the function is an array of BOOL.

If a result tag name is given in input, then this array inherits the number of elements from the tag itself. If no tag is given instead, this array has a number of elements equal to the number of fields of the targeted recipe + 3 (3 more fields for extra information).
The exact information stored in the returned array are:

1st element: a general flag, TRUE if a difference has been detected anywhere in the recipe;
2nd element: TRUE if a difference has been detected in the recipe NAME;
3rd element: TRUE if a difference has been detected in the recipe COMMENT;
next elements (one more element for each custom recipe field): TRUE if a difference has been detected in the corresponding custom recipe field.

If the number of array elements has been inherited from the provided result tag, then it is possible that this number doesn't match the exact need of the recipe; in this case the function ignores the mismatch:
- if there are more elements than needed, then the extra ones are left unaffected (FALSE);
- if there are less elements than needed, then there will simply be missing information in the result.

For example, in case of a recipe with 4 custom fields, the ideal number of array elements is 4 + 3 = 7;
- if a tag array with 10 elements is given, then the last 3 elements in the result will be unused;
- if a tag with 5 elements is given, then the flags for the last 2 recipe fields will be missing in the answer.

RECIPE_COMPAREFIELD

Check whether the value of a recipe field of a specific recipe already in archive is different from the value currently stored in its device tag.
This function automates the needed comparison activity, given a project that tries to keep the recipe comparison flags updated after changes in specific fields; the function implicitly updates the flags stored in the container tags automatically created by Crew™.
Similarly to the functions RECIPE_GETFIELDNAME and RECIPE_GETCOMPAREINDEX (see the related detailed descriptions), the recipe field can be identified by one of several different information:
- by its name (as configured in the project);
- by its index (the field positional index in the recipe structure, possible only with custom fields);
- by the ID of its device tag (possible only with fields that actually have one);
- by an address match between its associated device tag and a further given tag (meant to be used to match addresses with variable/variant tags, or in cases where different tags share the same address and the ID of the actual device tag is not known).

DIFFERENT = **RECIPE_COMPAREFIELD** (STRUCTURE, RECIPE, FIELD [, MODE])

input

STRUCTURE :	ANY_STRING	the name of the involved recipe structure
RECIPE :	ANY_STRING	the name of the recipe already in archive
FIELD :	ANY	a variable information used to identify the field; the meaning and type of this parameter depends on the value given to the (optional) 4th parameter (see below); the possibilities are: - MODE = **RECIPEBYINDEX** FIELD type = ANY_INT this FIELD parameter is the index (base-0) of the recipe field within the recipe structure; can range between 0 and RECIPE_GETFIELDSNUMBER - 1 this mode can only be used with custom fields, never with system fields - MODE = **RECIPEBYTAGID** FIELD type = UDINT this FIELD parameter is the ID of the device tag associated to the recipe field; the given ID is expected to be a valid tag ID for the current project; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) - MODE = **RECIPEBYTAGADD** FIELD type = UDINT this FIELD parameter is the ID of a tag to be used for an address comparison (the field is chosen if the address of its device tag is the same as the address of the tag given here); the given ID is expected to be a valid tag ID for the current project; it is usually (not necessarily) the ID of a variable/variant tag; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) - MODE = **RECIPEBYNAME** (also the **default**, in case an explicit MODE is missing) FIELD type = ANY_STRING this FIELD parameter is the name of the recipe field (the static name configured in the project); this mode can be used with either custom or system fields; in case of system fields names, the supported ones are:

- RECIPENAME ("RecipeName")
- RECIPECOMMENT ("Comment")

MODE :	ANY_INT	[OPTIONAL]	an identifier code defining the nature of the information given in the 2nd parameter, to be used to identify the recipe field (see above); the supported codes are: **0** (RECIPEBYINDEX) the field is identified by its index (base-0, relative to the recipe custom fields) **1** (RECIPEBYTAGID) the field is identified by the ID of its device tag (only possible for fields with a device tag) **2** (RECIPEBYTAGADD) the field is identified by the address of its device tag (only possible for fields with a device tag) **3** (RECIPEBYNAME) the field is identified by its name (this is the default as well: omitting this parameter means the field is identified by its preconfigured static name) if missing, the default behaviour is RECIPEBYNAME

output

DIFFERENT :	BOOL	the result of the comparison: TRUE means the archive and device values are different; FALSE means the two values are equal

This function will not simply check and return the difference between the archive and the device values:
the comparison result will also be automatically stored in the appropriate flag element of the tag-array configured as storage of the comparison flags. The 'generic' difference flag will be updated as well.

As also explained in the description of the functions RECIPE_GETFIELDNAME and RECIPE_GETCOMPAREINDEX, to further explain the behaviour of the function in case of RECIPEBYTAGADD mode:
the identified field is the one associated to a device tag that has the same address as the current address of the other (3rd parameter) identified tag;
the given tag can be a variable/variant tag, and is allowed to have a variable address; in this case the tag address should have been evaluated and assigned before the invocation of this function (that means the tag should have been read or written at least once before). The last evaluated address is matched against those of all the device tags associated to recipe fields, until a suitable correspondence is found.

example

```
// Setting a difference bit in a flags array (usage example for recipes comparison)
// 4. automatic mode: based on the address match of a field's tag

VAR CONSTANT
   _STRNAME : WSTRING [100] := "Recipe1";
   _RECNAME : WSTRING [100] := "a";
END_VAR;

FUNCTION CompareField
   VAR_INPUT
      tagid : UDINT;
   END_VAR;
   RECIPE_COMPAREFIELD (_STRNAME, _RECNAME, tagid, RECIPEBYTAGADD);
END_FUNCTION;
```

RECIPE_EXPORT

Exports in a CSV file the recipes of a given structure, or of all the structures.
Similar to **RECIPE_EXPORTFLAT**, but for ESA-formatted CSV files.

NUMREC = **RECIPE_EXPORT** (FILE [, STRUCTURE [, RECSLIST]])

input

FILE :	ANY_STRING		path and name of the exported file
STRUCTURE :	ANY_STRING	[OPTIONAL]	the name of the involved structure; if missing, then all the recipes of ALL the structures are exported together in the same file;
RECSLIST :	ANY	[OPTIONAL]	list of recipes to export; an optional parameter usable to specify the exact list of recipes that have to be included in the export; without this parameter, all the recipes in archive (optionally limited to the given structure) will be included; if this list is specified instead, then the recipes in archive will be included only if part of the list; note that this list can only be given if a specific structure is given as well (parameter STRUCTURE); the type of this parameter is formally specified as ANY, but the only allowed types are actually strings (either STRING or WSTRING, if one only recipe is required), or arrays of strings (of either type, if a whole set of recipes is required)

output

NUMREC :	UDINT	number of exported recipes

The function is meant to create CSV files in ESA-format
(a custom ESA layout, as opposed to flat CSV records; see **RECIPE_EXPORTFLAT** below).

In case the RECSLIST parameter is used to specify more than a single recipe (a whole list of recipes given as an array of strings), the array passed as parameter can either be exactly dimensioned to contain the precise number of required recipes, or even bigger. If bigger arrays are used, then an empty string must be assigned as an element, to mark the end of the list of names.

RECIPE_EXPORTFLAT

Exports in a CSV file the recipes of a given structure, or of all the structures.
Similar to **RECIPE_EXPORT**, but for flat CSV files.

NUMREC = **RECIPE_EXPORTFLAT** (FILE, STRUCTURE [, RECSLIST])

input			
FILE :	ANY_STRING		path and name of the exported file
STRUCTURE :	ANY_STRING		the name of the involved structure; unlike with the **RECIPE_EXPORT**, this parameter is mandatory: generic flat CSV files are not able to handle records of different recipes contemporarily;
RECSLIST :	ANY	[OPTIONAL]	list of recipes to export; an optional parameter usable to specify the exact list of recipes that have to be included in the export; without this parameter, all the recipes in archive (for the given structure) will be included; if this list is specified instead, then the recipes in archive will be included only if part of the list; the type of this parameter is formally specified as ANY, but the only allowed types are actually strings (either STRING or WSTRING, if one only recipe is required), or arrays of strings (of either type, if a whole set of recipes is required)

output		
NUMREC :	UDINT	number of exported recipes

The function is meant to create CSV files in a flat format, with semicolon-separated fields
(unlike the **RECIPE_EXPORT**, used to export CSV files with a custom ESA layout).

In case the RECSLIST parameter is used to specify more than a single recipe (a whole list of recipes given as an array of strings), the array passed as parameter can either be exactly dimensioned to contain the precise number of required recipes, or even bigger. If bigger arrays are used, then an empty string must be assigned as an element, to mark the end of the list of names.

RECIPE_IMPORT

Imports from a CSV file the recipes of a given structure, or of all the structures.

NUMREC = **RECIPE_IMPORT** (FILE [, STRUCTURE [, RECSLIST]])

input

FILE :	ANY_STRING		path and name of the exported file
STRUCTURE :	ANY_STRING	[OPTIONAL]	the name of the involved structure; if missing, then recipes of ALL the structures are imported from the same file (ESA-format only)
RECSLIST :	ANY	[OPTIONAL]	list of recipes to export; an optional parameter usable to specify the exact list of recipes that have to be included in the export; without this parameter, all the recipes in archive (optionally limited to the given structure) will be included; if this list is specified instead, then the recipes in archive will be included only if part of the list; note that this list can only be given if a specific structure is given as well (parameter STRUCTURE); the type of this parameter is formally specified as ANY, but the only allowed types are actually strings (either STRING or WSTRING, if one only recipe is required), or arrays of strings (of either type, if a whole set of recipes is required)

output

NUMREC :	UDINT	number of imported recipes

The function is automatically able to detect and handle files prepared with different encodings and layouts:
- ESA-format CSV files (can be UNICODE only),
- standard flat CSV, with semicolon-separated fields (either in UNICODE or ANSI),
- standard flat TXT, with TAB-separated fields (either in UNICODE or ANSI).

Note that files formatted with the ESA custom layout are able to contain definitions of recipes of different structures at once: the structure of each recipe is defined within the file. This means the programmer is allowed to use them with instructions meant to import the recipes of all of them contemporarily (or to choose a specific structure among them).
Flat CSV files, instead, don't contain information about the structure: programmers are forced to specify the structure the recipes are being imported for.

In case the RECSLIST parameter is used to specify more than a single recipe (a whole list of recipes given as an array of strings), the array passed as parameter can either be exactly dimensioned to contain the precise number of required recipes, or even bigger. If bigger arrays are used, then an empty string must be assigned as an element, to mark the end of the list of names.

When this function succeeds, two conventional variables can be used to retrieve additional information about the result:

RECIPE_IMPORTEDNEW gives the number of entirely new recipes, imported and added to the archive
RECIPE_IMPORTEDOLD gives the number of already known recipes, imported and replaced in the archive

In case of function failures, these variables are not affected (information from the last successful execution is retained).

example

```
VAR
   rcstr : WSTRING [64] := 'Recipe1';
   rnrec, rntot, rnadd, rnchg : UDINT;
END_VAR;

rnrec := RECIPE_GETNUMBER (rcstr);
rntot := RECIPE_IMPORT ('/mydocs/test-flat.csv', rcstr);  // must be <nradd> + <nrchg> from below
nradd := RECIPE_IMPORTEDNEW;
nrchg := RECIPE_IMPORTEDOLD;
rnrec := RECIPE_GETNUMBER (rcstr);                        // must be <rnrec> + <nradd> from above

rntot := RECIPE_IMPORT ('/mydocs/test-esa.csv');          // only ESA files usable for multiple structures
nradd := RECIPE_IMPORTEDNEW;
nrchg := RECIPE_IMPORTEDOLD;
```

RECIPE_PRINT

Prints the recipes of a given structure, or of all the structures.

NUMREC = **RECIPE_PRINT** ([STRUCTURE])

input

STRUCTURE :	ANY_STRING	[OPTIONAL]	the name of the involved structure; if missing, then the recipes of ALL the structures are printed

output

NUMREC :	UDINT	number of printed recipes

RECIPE_GETCURNAME

Retrieves the recipe name currently set in the structure buffer.

NAME = **RECIPE_GETCURNAME** (STRUCTURE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure

output		
NAME :	WSTRING	the current recipe name

The function simply returns the current value of the buffer tag associated to the recipe name.

RECIPE_EXIST

Checks whether a given recipe exists in a structure archive.

EXIST = **RECIPE_EXIST** (STRUCTURE, RECIPE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE :	ANY_STRING	the name of the involved recipe

output		
EXIST :	BOOL	TRUE if the specified recipe exists; FALSE otherwise

RECIPE_GETNUMBER

Retrieves the number of valid recipes currently stored in a structure archive.

NUMBER = **RECIPE_GETNUMBER** (STRUCTURE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure

input		
NUMBER :	UDINT	the number of used recipe records

See **RECIPE_DELETE** for notes regarding the management of deleted recipes and released records.

As per the explained mechanics, this function is used to retrieve the number of records actually used by existing recipes (in other words the number of recipes); that is not necessarily the total number of records in the archive (which is the aim of **RECIPE_GETRECORDS**).

example

```
VAR
   rcstr : WSTRING [64];
   rcnum : UDINT;
END_VAR;

rcstr := 'Recipe1';

rcstr := RECIPE_GETNUMBER (rcstr);     // = 0 (starting with an empty archive)
rcstr := RECIPE_GETRECORDS (rcstr);    // = 0 (starting with an empty archive)
RECIPE_SAVE (rcstr, 'AA');             // adding recipe AA
RECIPE_SAVE (rcstr, 'BB');             // adding recipe BB
RECIPE_SAVE (rcstr, 'CC');             // adding recipe CC
rcstr := RECIPE_GETNUMBER (rcstr);     // = 3 (counting all 3 used records)
rcstr := RECIPE_GETRECORDS (rcstr);    // = 3 (counting all 3 used records)
RECIPE_DELETE (rcstr, 'BB');           // deleting recipe BB only
rcstr := RECIPE_GETNUMBER (rcstr);     // = 2 (only AA and CC remaining)
rcstr := RECIPE_GETRECORDS (rcstr);    // = 3 (2 used records + 1 free reusable record)
RECIPE_DELETE (rcstr);                 // delete all the existing recipes
rcstr := RECIPE_GETNUMBER (rcstr);     // = 0 (no more recipes)
rcstr := RECIPE_GETRECORDS (rcstr);    // = 3 (3 free reusable records)
```

Further *examples* can be found throughout this document; see especially **RECIPE_PACKARCHIVE** and **RECIPE_SAVE**.

RECIPE_GETRECORDS

Retrieves the total number of the records taking up space in a structure archive.
The count includes both valid recipes records and empty records left by old removed recipes.

NUMBER = **RECIPE_GETRECORDS** (STRUCTURE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure

input		
NUMBER :	UDINT	the total number of used and unused records

See **RECIPE_DELETE** for notes regarding the management of deleted recipes and released records.

As per the explained mechanics, this function is used to retrieve the total number of records in the structure archive, including both used and free records; that is not simply the number of the existing recipes (which is the aim of **RECIPE_GETNUMBER**).

Several *example*s throughout this document are making use of this function; see especially **RECIPE_GETNUMBER**, **RECIPE_PACKARCHIVE**, **RECIPE_SAVE** and **RECIPE_GETINFO**.

RECIPE_GETINFO

Retrieves the name (and additional information) of a recipe with known ID.

NAME = **RECIPE_GETINFO** (STRUCTURE, ID)

input

STRUCTURE :	ANY_STRING	the name of the involved structure
ID :	UDINT	the ID (base-1 index) of the recipe

output

NAME :	WSTRING	the name of the referenced recipe

Provided the given ID is valid (within the archive range and pointing to a used record), the function immediately returns the name of the referenced recipe.
Further information though is stored in a set of dedicated variables:

RECIPE_NAME	replicates the name of the recipe, already returned by this function
RECIPE_ID	replicates the ID given to identify the recipe
RECIPE_COMMENT	the comment stored along with the recipe
RECIPE_TIME	the last save timestamp of the recipe

These variables are updated every time the **RECIPE_GETINFO** succeeds in retrieving a valid recipe and returning its name. Errors will not affect their values (**ERRNO** and empty NAME strings can be used to detect failure conditions): the information of the last successful execution is always retained.

This function is especially meant to be used in loops (where IDs spanning between 1 and **RECIPE_GETRECORDS** could be used); but it can even be used to retrieve information about known targeted recipes, provided their ID is known (see **RECIPE_GETID** as a method to retrieve IDs of recipes of known name).

example

```
VAR
   rcstr : WSTRING [64] := 'Recipe1';
   rname : WSTRING [64];
   rcnum , idx : UDINT;
   outstr : WSTRING [128];
END_VAR;

rcnum := RECIPE_GETRECORDS (rcstr);
ST_OPTION HANDLE_ERRORS;
ERRNO := 0;
FOR idx := 1 TO rcnum DO
   rname := RECIPE_GETINFO (rcstr, idx);
   IF ERRNO = 0 THEN
      outstr := rname+','+RECIPE_COMMENT+','+ANY_TO_STRING(RECIPE_ID)+','+ANY_TO_STRING(RECIPE_TIME);
   ELSE
      ERRNO := 0;
   END_IF;
END_FOR;
ST_OPTION BLOCKING_ERRORS;
```

A further *example* given below (see **RECIPE_GETID**) might be significant.

RECIPE_GETID

Retrieves the ID of a recipe with a given name.

ID = **RECIPE_GETID** (STRUCTURE, RECIPE)

input

STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE :	ANY_STRING	the name of the involved recipe

output

ID :	UDINT	the ID of the recipe; this ID is defined as the index (base-1) of the recipe within the archive

example

```
VAR
   rcstr : WSTRING [64];
   rname1, rname2 : WSTRING [64];
   rcidx : UDINT;
   outstr : WSTRING [128];
END_VAR;

rcstr := 'Recipe1';

rname1 := RECIPE_GETCURNAME (rcstr);
rcidx  := RECIPE_GETID (rcstr, rname1);
rname2 := RECIPE_GETINFO (rcstr, rcidx);
outstr := 'name :' + RECIPE_NAME;  // = rname1, = rname2
outstr := 'comment :' + RECIPE_COMMENT;
outstr := 'id :' + ANY_TO_STRING (RECIPE_ID);
outstr := 'time :' + ANY_TO_STRING (RECIPE_TIME);
```

RECIPE_GETFIELDSNUMBER

Retrieves the number of custom fields configured in a recipe structure (system fields such as name, comment, time and ID are excluded from the count).

FIELDS = **RECIPE_GETFIELDSNUMBER** (STRUCTURE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure

output		
FIELDS :	UDINT	the number of custom fields

RECIPE_GETFIELDNAME

Retrieves the name of one of the fields of a recipe structure.
Similarly to the functions RECIPE_COMPAREFIELD and RECIPE_GETCOMPAREINDEX (see the related detailed descriptions), the recipe field can be identified by one of several different information:
- by its index (the field positional index in the recipe structure);
- by the ID of its device tag (possible only for fields that actually have one);
- by an address match between its associated device tag and a further given tag (meant to be used to match addresses with variable/variant tags, or in cases where different tags share the same address and the ID of the actual device tag is not known).

NAME = **RECIPE_GETFIELDNAME** (STRUCTURE, FIELD [, MODE])

input

STRUCTURE :	ANY_STRING		the name of the involved structure
FIELD :	ANY_INT		a variable information used to identify the field; the meaning and type of this parameter depends on the value given to the (optional) 3rd parameter (see below); the possibilities are: - MODE = **RECIPEBYINDEX** (also the **default**, in case an explicit MODE is missing) FIELD type = ANY_INT this FIELD parameter is the index (base-0) of the recipe field within the recipe structure; can range between 0 and RECIPE_GETFIELDSNUMBER - 1 this mode can only be used with custom fields, never with system fields - MODE = **RECIPEBYTAGID** FIELD type = UDINT this FIELD parameter is the ID of the device tag associated to the recipe field; the given ID is expected to be a valid tag ID for the current project; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) [→ RECIPE_GETFIELDNAMEDTAG] - MODE = **RECIPEBYTAGADD** FIELD type = UDINT this FIELD parameter is the ID of a tag to be used for an address comparison (the field is chosen if the address of its device tag is the same as the address of the tag given here); the given ID is expected to be a valid tag ID for the current project; it is usually (not necessarily) the ID of a variable/variant tag; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) [→ RECIPE_GETFIELDNAMEDADD]
MODE :	ANY_INT	[OPTIONAL]	an identifier code defining the nature of the information given in the 2nd parameter, to be used to identify the recipe field (see above); the supported codes are: **0** (RECIPEBYINDEX) the field is identified by its index (this is the default as well: omitting this parameter means the field is identified by its index) **1** (RECIPEBYTAGID) the field is identified by the ID of its device tag (only possible for fields with a device tag) **2** (RECIPEBYTAGADD) the field is identified by the address of its device tag (only possible for fields with a device tag)

if missing, the default behaviour is **RECIPEBYINDEX**

output

NAME :	WSTRING	name of the requested field; note that if the selected field is one of the 4 system fields, this function could even return one of the supported system names: - **RECIPENAME** ("RecipeName") - **RECIPEID** ("RecipeId") - **RECIPECOMMENT** ("Comment") - **RECIPETIME** ("ChangeTime")

As also explained in the description of the functions **RECIPE_COMPAREFIELD** and **RECIPE_GETCOMPAREINDEX**, to further explain the behaviour of the function in case of **RECIPEBYTAGADD** mode:
the returned field is the one associated to a device tag that has the same address as the current address of the other (2nd parameter) identified tag;
the given tag can be a variable/variant tag, and is allowed to have a variable address; in this case the tag address should have been evaluated and assigned before the invocation of this function (that means the tag should have been read or written at least once before). The last evaluated address is matched against those of all the device tags associated to recipe fields, until a suitable correspondence is found.

example

```
VAR
   FieldsNumber : UDINT;
   FieldName : WSTRING [128];
   idx : UDINT;
END_VAR;

FieldsNumber := RECIPE_GETFIELDSNUMBER ("Recipe1");
FOR idx := 0 TO FieldsNumber-1 DO
   FieldName := RECIPE_GETFIELDNAME ("Recipe1", idx);
   _TRACE (ANY_TO_WSTRING(idx) + ") " + FieldName);
END_FOR;
```

example

```
// Setting a difference bit in a flags array (usage example for recipes comparison)
// 2. comprehensive mode: based on the address match of a field's tag

// Environment constants
VAR CONSTANT
   _STRNAME : WSTRING [100] := "Recipe1";
   _RECNAME : WSTRING [100] := "a";
   _TAGDIFF : WSTRING [100] := "Tag_DIFF";
END_VAR;

// Execute the field value comparison (archive/device)
FUNCTION CompareField

   VAR_INPUT
      tagid : UDINT;                                          // - tag ID given in input
   END_VAR;
   VAR
      fieldname : WSTRING [256];
      different : BOOL;
   END_VAR;

   fieldname := RECIPE_GETFIELDNAME (_STRNAME, tagid, RECIPEBYTAGADD);     // - identify the recipe field
   different := ( RECIPE_GETFIELDVALUE (_STRNAME, _RECNAME, fieldname) <>  // - compare archive value
                  TAG_READVALUE (TAG_GETNAME (tagid)) );                   //   and device tag
   SetDiffResult (fieldname, different);                                   // - apply comparison result
```

```
END_FUNCTION;

// Set up the proper flags, given a field state
FUNCTION SetDiffResult

   VAR_INPUT
      fieldname : WSTRING [256];                                    // - recipe field name
      different : BOOL;                                             // - difference flag
   END_VAR;
   VAR
      index : UDINT;
      flags : ARRAY [256] OF BOOL;
   END_VAR;

   // Find field index
   IF (fieldname = RECIPENAME) THEN
      index := RECIPECOMPNAME;                                      // - fixed index for name field
   ELSIF (fieldname = RECIPECOMMENT) THEN
      index := RECIPECOMPCOMMENT;                                   // - fixed index for comment field
   ELSE
      index := RECIPECOMPCUSTOM + RECIPE_GETFIELDINDEX (_STRNAME, fieldname);  // - custom field index
   END_IF;

   // Set field flag
   TAG_WRITEELEMENT (_TAGDIFF, index, different);                   // - store flag in tag array

   // Set global flag
   IF (different) THEN
      TAG_WRITEELEMENT (_TAGDIFF, 0, TRUE);
   ELSE
      ST_OPTION LAX_TYPES;
      flags := TAG_GETVALUE (_TAGDIFF);
      ST_OPTION STRICT_TYPES;
      FOR index := 1 TO RECIPE_GETFIELDSNUMBER(_STRNAME)+2 DO
         IF (flags[index]) THEN
            different := TRUE;
            exit;
         END_IF;
      END_FOR;
      TAG_WRITEELEMENT (_TAGDIFF, RECIPECOMPGENERAL, different);
   END_IF;

END_FUNCTION;
```

RECIPE_GETFIELDINDEX

Retrieves the index (base-0) of a given custom recipe field.

INDEX = **RECIPE_GETFIELDINDEX** (STRUCTURE, FIELD)

input

STRUCTURE :	ANY_STRING	the name of the involved structure
FIELD :	ANY_STRING	the name of the requested field; expected to be the name of one of the custom fields of the given structure (system fields such as recipe name, comment, ID and time are excluded)

output

INDEX :	UDINT	the index (base-0) of the needed field can range between 0 and RECIPE_GETFIELDSNUMBER - 1

example

```
// Setting a difference bit in a flags array (usage example for recipes comparison)
// 1. basic mode: based on the positional index of a recipe field
//    (needs knowledge of field name, value and type)

FUNCTION CompareFieldNumeric                                    // Execute the (numeric) comparison
   VAR_INPUT
      fieldname : WSTRING [256];                                // - recipe field name
      fieldvalue : UINT;                                        // - recipe field new value
   END_VAR;
   VAR
      archivevalue : UINT;
      different : BOOL;
   END_VAR;
   archivevalue := RECIPE_GETFIELDVALUE ("Recipe1", "a", fieldname);  // - get old value from archive
   different := (archivevalue <> fieldvalue);                   // - check new/old values difference
   SetDiffResult (fieldname, different);                        // - set up the comparison reult
END_FUNCTION;

FUNCTION SetDiffResult                                          // Set the proper flag
   VAR_INPUT
      fieldname : WSTRING [256];                                // - recipe field name
      different : BOOL;                                         // - difference flag
   END_VAR;
   VAR
      index : UDINT;
   END_VAR;
   IF (fieldname = RECIPENAME) THEN
      index := RECIPECOMPNAME;                                  // - fixed index for name field
   ELSIF (fieldname = RECIPECOMMENT) THEN
      index := RECIPECOMPCOMMENT;                               // - fixed index for comment field
   ELSE
      index := RECIPECOMPGENERAL + RECIPE_GETFIELDINDEX ("Recipe1", fieldname);  // - custom field index
   END_IF;
   _TRACE (fieldname + "(" + ANY_TO_STRING(index) + ") : " + SEL (different, "Equal", "Different "));
   TAG_WRITEELEMENT ("Tag_DIFF", index, different);             // - store flag in tag array
END_FUNCTION;
```

RECIPE_GETCOMPAREINDEX

Retrieves the index of a recipe comparison flag.
The retrieved index is referred to the element of the comparison array prepared by a RECIPE_COMPARESET (or equivalent), associated to the specified recipe field.
Similarly to the functions RECIPE_COMPAREFIELD and RECIPE_GETFIELDNAME (see the related detailed descriptions), the recipe field can be identified by one of several different information:
- by its name (as configured in the project);
- by its index (the field positional index in the recipe structure);
- by the ID of its device tag (possible only for fields that actually have one);
- by an address match between its associated device tag and a further given tag (meant to be used to match addresses with variable/variant tags, or in cases where different tags share the same address and the ID of the actual device tag is not known).

In few words, this is the function to use when the script needs to know the element of the comparison array where is stored the flag of a specific recipe field.

INDEX = **RECIPE_GETCOMPAREINDEX** (STRUCTURE, FIELD [, MODE])

input

STRUCTURE :	ANY_STRING	the name of the involved recipe structure
FIELD :	ANY	a variable information used to identify the field; the meaning and type of this parameter depends on the value given to the (optional) 3rd parameter (see below); the possibilities are: - MODE = **RECIPEBYINDEX** FIELD type = ANY_INT this FIELD parameter is the index (base-0) of the recipe field within the recipe structure; can range between 0 and RECIPE_GETFIELDSNUMBER - 1 this mode can only be used with custom fields, never with system fields - MODE = **RECIPEBYTAGID** FIELD type = UDINT this FIELD parameter is the ID of the device tag associated to the recipe field; the given ID is expected to be a valid tag ID for the current project; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) - MODE = **RECIPEBYTAGADD** FIELD type = UDINT this FIELD parameter is the ID of a tag to be used for an address comparison (the field is chosen if the address of its device tag is the same as the address of the tag given here); the given ID is expected to be a valid tag ID for the current project; it is usually (not necessarily) the ID of a variable/variant tag; this mode can only be used with fields that actually have a device tag association (all the custom fields and only the system fields with an explicit device association) - MODE = **RECIPEBYNAME** (also the **default**, in case an explicit MODE is missing) FIELD type = ANY_STRING this FIELD parameter is the name of the recipe field (the static name configured in the project); this mode can be used with either custom or system fields; in case of system fields names, the supported ones are: - **RECIPEGENERAL** ("RecipeGeneral", for the overall comparison flag)

- RECIPENAME ("RecipeName")
- RECIPECOMMENT ("Comment")

MODE :	ANY_INT	[OPTIONAL]	an identifier code defining the nature of the information given in the 2nd parameter, to be used to identify the recipe field (see above); the supported codes are: **0** (RECIPEBYINDEX) the field is identified by its index (base-0, relative to the recipe custom fields) **1** (RECIPEBYTAGID) the field is identified by the ID of its device tag (only possible for fields with a device tag) **2** (RECIPEBYTAGADD) the field is identified by the address of its device tag (only possible for fields with a device tag) **3** (RECIPEBYNAME) the field is identified by its name (this is the default as well: omitting this parameter means the field is identified by its preconfigured static name) if missing, the default behaviour is RECIPEBYNAME

output

INDEX :	UDINT	index of the array element containing the requested field flag; if the selected field is one of the 3 supported system elements, the returned indexes should be: - RECIPEGENERAL ("RecipeGeneral") → 0 - RECIPENAME ("RecipeName") → 1 - RECIPECOMMENT ("Comment") → 2 custom fields should have flags starting from index 3

To better handle the indexes of system and custom elements, the programmer can use the following reserved symbolics as numeric constants:
RECIPECOMPGENERAL : 0 (the index of the general flag)
RECIPECOMPNAME : 1 (the index of the recipe name flag)
RECIPECOMPCOMMENT : 2 (the index of the recipe comment flag)
RECIPECOMPCUSTOM : 3 (the index of the first custom field flag)

As also explained in the description of the functions RECIPE_COMPAREFIELD and RECIPE_GETFIELDNAME, to further explain the behaviour of the function in case of RECIPEBYTAGADD mode:
the identified field is the one associated to a device tag that has the same address as the current address of the other (2nd parameter) identified tag;
the given tag can be a variable/variant tag, and is allowed to have a variable address; in this case the tag address should have been evaluated and assigned before the invocation of this function (that means the tag should have been read or written at least once before). The last evaluated address is matched against those of all the device tags associated to recipe fields, until a suitable correspondence is found.

example

```
// Setting a difference bit in a flags array (usage example for recipes comparison)
// 3. comprehensive mode: based on the address match of a field's tag (with flag index identification)

// Environment constants
VAR CONSTANT
   _STRNAME : WSTRING [100] := "Recipe1";
   _RECNAME : WSTRING [100] := "a";
   _TAGDIFF : WSTRING [100] := "Tag_DIFF";
END_VAR;

// Execute the field value comparison (archive/device)
FUNCTION CompareField

   VAR_INPUT
```

```
      tagid : UDINT;                                                    // - tag ID given in input
   END_VAR;
   VAR
      fieldname : WSTRING [256];
      different : BOOL;
   END_VAR;

   fieldname := RECIPE_GETFIELDNAME (_STRNAME, tagid, RECIPEBYTAGADD);     // - identify the recipe field
   different := ( RECIPE_GETFIELDVALUE (_STRNAME, _RECNAME, fieldname) <>  // - compare archive value
                  TAG_READVALUE (TAG_GETNAME (tagid)) );                   //   and device tag
   SetDiffResult (fieldname, different);                                   // - apply comparison result

END_FUNCTION;

// Set up the proper flags, given a field state
FUNCTION SetDiffResult

   VAR_INPUT
      fieldname : WSTRING [256];                                        // - recipe field name
      different : BOOL;                                                 // - difference flag
   END_VAR;
   VAR
      index : UDINT;
      flags : ARRAY [256] OF BOOL;
   END_VAR;

   // Find field index
   index := RECIPE_GETCOMPAREINDEX (_STRNAME, fieldname, RECIPEBYNAME);  // - find field index

   // Set field flag
   TAG_WRITEELEMENT (_TAGDIFF, index, different);                       // - store flag in tag array

   // Set global flag
   IF (different) THEN
      TAG_WRITEELEMENT (_TAGDIFF, 0, TRUE);
   ELSE
      ST_OPTION LAX_TYPES;
      flags := TAG_GETVALUE (_TAGDIFF);
      ST_OPTION STRICT_TYPES;
      FOR index := 1 TO (RECIPE_GETFIELDSNUMBER(_STRNAME) + RECIPECOMPCUSTOM - 1) DO
         IF (flags[index]) THEN
            different := TRUE;
            exit;
         END_IF;
      END_FOR;
      TAG_WRITEELEMENT (_TAGDIFF, RECIPECOMPGENERAL, different);
   END_IF;

END_FUNCTION;
```

RECIPE_GETFIELDVALUE

Retrieves from the archive the value of a field of a given recipe.

VALUE = **RECIPE_GETFIELDVALUE** (STRUCTURE, RECIPE, FIELD)

input

STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE :	ANY_STRING	the name of the involved recipe
FIELD :	ANY_STRING	the name of the involved field; this could be either: - the user-defined name of any structure field - "RecipeName" (or **RECIPENAME**) - "RecipeId" (or **RECIPEID**) - "Comment" (or **RECIPECOMMENT**) - "ChangeDate" or "ChangeTime" (or **RECIPETIME**)

output

VALUE :	ANY	the value, retrieved from the archive, of the identified recipe field; the type of the returned value will match that of the configured field; in case of one of the special fields, the expected return type is: **RECIPENAME** : WSTRING **RECIPEID** : UDINT **RECIPECOMMENT** : WSTRING **RECIPETIME** : LDT

This function is used to access single pieces of a recipe directly from the archive, without affecting the current content of the recipe buffer.

RECIPE_SETFIELDVALUE

Writes in the archive the value of a field of a given recipe.

RECIPE_SETFIELDVALUE (STRUCTURE, RECIPE, FIELD, VALUE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE :	ANY_STRING	the name of the involved recipe
FIELD :	ANY_STRING	the name of the involved field; this could be either: - the user-defined name of any structure field - "Comment" (or **RECIPECOMMENT**) - "ChangeDate" or "ChangeTime" (or **RECIPETIME**) note that Name and ID of a recipe already in archive can't be changed by simply accessing the information (these are system information only affected by specific recipe-wide operations, such as whole recipes saving, renaming, and archive packing operations)
VALUE :	ANY	the new value of the identified recipe field, to be written in the archive; the type of the given value must match that of the configured field; in case of one of the special fields, the expected value type is: **RECIPECOMMENT** : WSTRING **RECIPETIME** : LDT

This function is used to access single pieces of a recipe directly in the archive, without the need to affect the current content of the recipe buffer.

Note that the submitted changes are directly and silently applied to the content of the interested recipe archive, no automatic reload of tags , buffers or client pages is implied by this operation.

RECIPE_SETFIELDEXPORT

Set the export format of specific recipe fields.

RECIPE_SETFIELDEXPORT (STRUCTURE, FIELD, FORMAT)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure
FIELD :	ANY_STRING	the name of the recipe field that has to be redefined; this parameter is expected to identify one of the custom-defined fields; format changes on system fields are not allowed
FORMAT :	ANY_STRING	an identifier of the new format that has to be applied to the field; format codes currently supported are: - "**DT**" : can be applied to numeric integer values expected to have date/time values in DT format (the numeric equivalent is a 32bit integer value with the number of seconds elapsed since 01/01/1970); the export will convert the numeric value in the corresponding date/time string; - "**LDT**" : can be applied to numeric integer values expected to have date/time values in LDT format (the numeric equivalent is a 64bit integer value with the number of 100-nanoseconds elapsed since 01/01/1601); the export will convert the numeric value in the corresponding date/time string; - "**DT_m**" : is a custom format that can be applied to numeric integer values expected to have date/time values expressed in «milliseconds elapsed since 01/01/1970-00:00:00»; the export will convert the numeric value in the corresponding date/time string; - "**.##**" : a dot followed by numeric digits is used to specify the number of decimal digits to be used with floating point values; only values in the range 0..99 are allowed - "" : an empty string is used as a special directive meant to reset the field format to its default

This function is used to change the format of given recipe fields when exported in a file.
The selection will have effect on both 'ESA' and 'flat' export formats.

Note that the format redefinition applied by this function has only effect on exports toward files, not on exports on database, since the format of database columns can't be freely changed at runtime.
Also note that files exported using redefined formats might no longer be compatible for imports (or might lead to unexpected values), if the change goes against the fields basic nature. For example, forcing only fields decimals precisions won't affect compatibility for imports, while changing a numeric value in a date/time string will.

Remember that it is possible to reset the format to the field default simply passing an empty FORMAT string to this function.

RECIPE_GETSTRRECORD

Retrieves a recipe record from the archive.

RECORD = **RECIPE_GETSTRRECORD** (STRUCTURE, RECIPE, TYPE)

input		
STRUCTURE :	ANY_STRING	the name of the involved structure
RECIPE :	ANY	the recipe to be retrieved; - if given as a string (ANY_STRING), it must be the name of the needed recipe; - if given as a numeric value (ANY_INT), it must be the ID (base-1 index) of the recipe; can be in the range between 1 and RECIPE_GETRECORDS; check the function result (in HANDLE_ERRORS mode), or the returned recipe name, to see whether the given parameter identified a valid recipe
TYPE :	ANY_STRING	the name of a user-defined structured type, resembling the layout of the structure of the recipe (note that only structures are allowed); this can be a type manually prepared by the ST programmer, or - ideally - automatically predefined in the system ST section by the configurator compiler
output		
RECORD :	ANY	the returned value assumes the (structured) type with the name given in the parameter TYPE; this structure fields are filled up with the values of the fields of the identified recipe, as currently stored in the archive

The function is meant to return a whole recipe record acquired from the archive.

The ST programmer (or the compilation framework) must have created a user-defined (structured) type resembling the layout of the recipe structure. This type must be specified to the function, and is used to give form to the returned value.
The given user-defined type structure, and the recipe structure, are not required to be perfectly identical. In fact, this function could be used to retrieve even a subset of recipe fields, if a properly defined destination structured type is provided.

Assignment rules:
- assignments are done field by field;
- fields match is done by name;
  the names in the ST structure should match the names of the recipe items;
  custom items have their own user-defined names, while the system fields are expected to have the following conventional names:
  . “RecId” (the recipe ID, should be a UDINT),
  . “RecName” (the recipe name, should be a WSTRING[32]),
  . “RecTime” (the last change time, should be an LDT, even though natively stored as a plain ‘ulong’),
  . “RecComment” (the recipe comment, should be a WSTRING[100]);
- fields that exist in the source recipe record (as defined in the recipe structure), but not in the destination ST value (as defined in the ST user-defined type), are ignored;
- fields that exist in the destination ST value, but not in the source recipe record, are explicitly reset in the destination;
- where the names match is verified, the fields types are required to match as well; perfection is not required, but checks are strict:
  . integers can only be assigned to integers (or pseudo-integers) of the same size
    (pseudo-integers definition includes integers, ranges, enumeratives, bitstrings, booleans, date/time),

- reals can only be assigned to reals of the same size,
- strings can only be assigned to strings of the same size and type,
- arrays can only be assigned to arrays of the same size and with elements following the rules above,
- sub-structures and arrays of sub-structures (meaning fields of structured type) are transferred as binary values; they are required to perfectly match in definition (or at least to match the rules above field by field, so that the overall binary layout of the two parts is identical).

RECIPE_GETSTRRECORDS

Retrieves a set of recipe records from the archive.

RECORDS = **RECIPE_GETSTRRECORDS** (STRUCTURE, TYPE, SIZE, PAGE [, FROM, TO])

input			
STRUCTURE :	ANY_STRING		the name of the involved structure
TYPE :	ANY_STRING		the name of a user-defined structured type, resembling the layout of the structure of the recipe (note that only structures are allowed); this can be a type manually prepared by the ST programmer, or - ideally - automatically predefined in the system ST section by the configurator compiler
SIZE :	ANY_INT		maximum number of recipes returned by the function (and number of elements of the returned array of structures)
PAGE :	ANY_INT		in case too many recipe records match the request, and only a part of them can be returned (SIZE), this parameter can be used to choose which of them are required (the 1st N, or the 2nd N, etc.); the 1st set of SIZE records is PAGE 1
FROM :	LDT	[OPTIONAL]	the timestamp of the first (oldest) recipe to be retrieved; if all records are needed and no timestamp range is required, this parameter can be omitted, or explicitly given as **ANY_TO_LDT** (0)
TO :	LDT	[OPTIONAL]	the timestamp of the last (newest) recipe to be retrieved if all records are needed and no timestamp range is required, this parameter can be omitted, or explicitly given as **ANY_TO_LDT** (-1)

output		
RECORDS :	ANY	an array of structures, meant to contain the retrieved recipes data; the returned array elements assume the (structured) type with the name given in the parameter TYPE; the array will have exactly the size (number of elements) specified in the SIZE parameter, even if less recipes were found (extra elements will have empty values); the elements structure fields are filled up with the values of the fields of the identified recipes, as currently stored in the archive

The function is meant to return a set of recipe records acquired from the archive.

The ST programmer (or the compilation framework) must have created a user-defined (structured) type resembling the layout of the recipe structure. This type must be specified to the function, and is used to give form to the returned value.
The given user-defined type structure, and the recipe structure, are not required to be perfectly identical. In fact, this function could be used to retrieve even a subset of recipe fields, if a properly defined destination structured type is provided.

See the specifications of the **RECIPE_GETSTRRECORD** function above for a description of the structures assignments rules.

RECIPE_GETTAGNAME

Retrieves the name of the tag associated to a specific structure field.

TAG = **RECIPE_GETTAGNAME** (STRUCTURE, FIELD, DEVICE)

input

STRUCTURE :	ANY_STRING	the name of the involved structure
FIELD :	ANY_STRING	the name of the involved field; this could be either: - the user-defined name of any structure field - "RecipeName" (or **RECIPENAME**) - "RecipeId" (or **RECIPEID**) - "Comment" (or **RECIPECOMMENT**) - "ChangeDate" or "ChangeTime" (or **RECIPETIME**) - "CompareArray" (or **RECIPECOMPAREA**) - "CompareStructure" (or **RECIPECOMPARES**)
DEVICE :	BOOL	TRUE to retrieve the device tag name; FALSE to retrieve the buffer tag name; note that this flag is ignored in case the request is done for the **RECIPECOMPAREA** or **RECIPECOMPARES** elements (not strictly related to recipe fields, rather to structure-related tags)

output

TAG :	WSTRING	the name of the tag

example

```
VAR
   rctndev  : WSTRING [64];
   rctnbuf  : WSTRING [64];
   rcstr    : WSTRING [64];
   rcfld    : WSTRING [64];
END_VAR;

rcstr := 'Recipe1';

rctndev := RECIPE_GETTAGNAME (rcstr, RECIPENAME   , TRUE );
rctnbuf := RECIPE_GETTAGNAME (rcstr, RECIPENAME   , FALSE);
// ... do whatever needed with the tags names
rctndev := RECIPE_GETTAGNAME (rcstr, RECIPEID     , TRUE );
rctnbuf := RECIPE_GETTAGNAME (rcstr, RECIPECOMMENT, FALSE);
rctndev := RECIPE_GETTAGNAME (rcstr, RECIPETIME   , TRUE );

rcfld := 'RecField1';
rctndev := RECIPE_GETTAGNAME (rcstr, rcfld, TRUE );
rctnbuf := RECIPE_GETTAGNAME (rcstr, rcfld, FALSE);
// ... do whatever needed with the tags names
rcfld := 'RecField2';
rctndev := RECIPE_GETTAGNAME (rcstr, rcfld, TRUE );
rctnbuf := RECIPE_GETTAGNAME (rcstr, rcfld, FALSE);
rcfld := 'RecField3';
rctndev := RECIPE_GETTAGNAME (rcstr, rcfld, TRUE );
rctnbuf := RECIPE_GETTAGNAME (rcstr, rcfld, FALSE);
```

< VARIABLES >

The following are the variables usable to share information and directives for the recipes management:

RECIPE_IMPORTEDNEW
type UDINT
access R
gives the number of imported recipes that were entirely unknown and added to the archive by a successful execution of RECIPE_IMPORT

RECIPE_IMPORTEDOLD
type UDINT
access R
gives the number of imported recipes that were already known and replaced in the archive by a successful execution of RECIPE_IMPORT

RECIPE_NAME
type WSTRING
access R
replicates the name of the last recipe retrieved by a successful execution of RECIPE_INFO (the same name already returned by it)

RECIPE_ID
type UDINT
access R
replicates the ID of the last recipe retrieved by a successful execution of RECIPE_INFO (the same ID passed as parameter to it)

RECIPE_COMMENT
type WSTRING
access R
gives the comment stored within the last recipe retrieved by a successful execution of RECIPE_INFO

RECIPE_TIME
type LDT
access R
gives the date and time of the last change/save operation made on the last recipe retrieved by a successful execution of RECIPE_INFO

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
RECIPENAME	"RecipeName"	**RECIPE_GETTAGNAME**, -**_GETFIELDNAME**, -**_GETFIELDVALUE**, …
RECIPEID	"RecipeId"	**RECIPE_GETTAGNAME**, -**_GETFIELDNAME**, -**_GETFIELDVALUE**
RECIPECOMMENT	"Comment"	**RECIPE_GETTAGNAME**, -**_GETFIELDNAME**, -**_GETFIELDVALUE**|**SET**, …
RECIPETIME	"ChangeTime"	**RECIPE_GETTAGNAME**, -**_GETFIELDNAME**, -**_GETFIELDVALUE**|**SET**
RECIPECOMPAREA	"CompareArray"	**RECIPE_GETTAGNAME**
RECIPECOMPARES	"CompareStructure"	**RECIPE_GETTAGNAME**
RECIPEGENERAL	"RecipeGeneral"	**RECIPE_GETCOMPAREINDEX**
RECIPEBYINDEX	0	**RECIPE_GETCOMPAREINDEX**, **RECIPE_GETFIELDNAME**
RECIPEBYTAGID	1	**RECIPE_GETCOMPAREINDEX**, **RECIPE_GETFIELDNAME**
RECIPEBYTAGADD	2	**RECIPE_GETCOMPAREINDEX**, **RECIPE_GETFIELDNAME**
RECIPEBYNAME	3	**RECIPE_GETCOMPAREINDEX**
RECIPECOMPGENERAL	0	**RECIPE_GETCOMPAREINDEX**
RECIPECOMPNAME	1	**RECIPE_GETCOMPAREINDEX**
RECIPECOMPCOMMENT	2	**RECIPE_GETCOMPAREINDEX**
RECIPECOMPCUSTOM	3	**RECIPE_GETCOMPAREINDEX**

15. RUNTIME - SAMPLES

DLOG_ENABLE

Enables the activity of a datalog samples buffer.

DLOG_ENABLE (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

DLOG_DISABLE

Disables the activity of a datalog samples buffer.

DLOG_DISABLE (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

DLOG_RESETSAMPLES

Cleans up the content of a datalog samples buffer.

DLOG_RESETSAMPLES (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

After the call, the datalog buffer will result completely empty.

DLOG_ACQUIRESAMPLES

Starts the acquisition of a set of samples of a datalog buffer.

DLOG_ACQUIRESAMPLES (DLOG)

input

DLOG :	ANY_STRING	the name of the datalog buffer

The acquisition will affect all the sources included in the datalog buffer.
Note that in case of sources based on tag-arrays, a single acquisition request will cause the creation of several samples contemporarily: one for each element of the acquired arrays.

The acquisition operation involves communication with external devices and is treated asynchronously; this function only triggers the start of the operation.
It is not possible to request acquisitions for a buffer if there is already another acquisition in progress (for the same buffer); remember though that if not configured properly, different buffers may fall in the same optimized acquisition group and this sort of acquisition 'busy' state will affect all of them contemporarily.

To minimize potential programming issues arising from the asynchronous behaviour of the acquisitions, a couple of utility functions are available to handle their states; see **DLOG_ACQUISITIONBUSY** and **DLOG_ACQUISITIONWAIT**.

example

```
VAR
   dlname : STRING [32];
   dlnum : UDINT;
   dlidx : UDINT;
END_VAR;

dlname := 'DataLog1';

FOR dlidx := 1 TO 100 DO

   DLOG_ACQUIRESAMPLES (dlname);             // start the acquisition process
   dlnum := DLOG_GETNUMSAMPLES (dlname);     // here the samples counter is still unchanged

   // Waiting mode 1: plain SLEEPs are not efficient at all
   // SLEEP (50);

   // Waiting mode 2: the BUSY flag allows even to handle waiting mechanics
   // WHILE DLOG_ACQUISITIONBUSY (dlname) DO
   //    SLEEP (1);
   // END_WHILE;

   // Waiting mode 3: the most efficient way to wait
   DLOG_ACQUISITIONWAIT (dlname);

   dlnum := DLOG_GETNUMSAMPLES (dlname);     // here the samples counter is finally updated

END_FOR;
```

DLOG_ACQUISITIONBUSY

Checks whether a samples acquisition is in progress for a given datalog.

STATE = **DLOG_ACQUISITIONBUSY** (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

output		
STATE :	BOOL	TRUE if there is an acquisition in progress; FALSE otherwise

See **DLOG_ACQUIRESAMPLES** for notes and usage examples.

DLOG_ACQUISITIONWAIT

Waits for a samples acquisition to complete for a given datalog.

STATE = **DLOG_ACQUISITIONWAIT** (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

output		
STATE :	BOOL	TRUE if an acquisition in progress had actually to be waited for; FALSE if there was nothing to wait for

This function allows the script to wait for the whole duration of an ongoing acquisition. While the script is waiting no other script can be executed, but further predefined functions could happen and the terminal would not feel blocked.
Nothing happens - not even errors notifications - if there is no acquisition currently in progress; see the returned state.

See **DLOG_ACQUIRESAMPLES** for notes and usage examples.

DLOG_APPENDSAMPLES

Appends a new set of samples to the bottom of the buffers of the sources of a datalog.

DLOG_APPENDSAMPLES (DLOG, VALUE1 [, VALUE2 [, ...]])

input

DLOG :	ANY_STRING	the name of the datalog buffer
VALUE# :	ANY_ELEMENTARY	a list of parameters containing the values of the new samples; in this list of values there MUST be exactly a value for each source of the datalog; the values MUST be given in the same number, order and types of the configured sources; currently, up to 15 values can be given, being 16 the maximum number of parameters for a script function; this means that datalogs with more than 15 sources can't be used with this function; most of the plain numeric types are acceptable for samples of numeric sources (basically, everything except strings, dates and times; values are ultimately converted in LREAL); string values are acceptable for samples of string sources

Programmers should be careful not to mess with buffers that are being changed by automatic system activities: adding custom samples to the buffers can lead to incoherent datalogs if not done properly.
The usage of this directive is recommended only with datalogs in manual mode, while no further acquisition from devices is in progress.

When appended, the current date and time is used as timestamp for the samples.

The given samples must be coherent with the configuration of the referenced datalog:
when part of a datalog, all the sources MUST share the same size, the same number of samples, all with corresponding timestamps; for this reason, appending samples to a single source is not allowed: samples for all the datalog sources must be given contemporarily, to ensure a continuous validity of the buffers;
the programmer must be sure to pass to the function the correct number of samples in the correct order (the parameters must match the number and order of the configured sources);
also the types of the values matter: the values types must match the type of the sources; any plain numeric value is acceptable for numeric sources, and any string is acceptable for string sources.

DLOG_FLUSH

Flushes on persistent storage the samples collected for a datalog buffer.

DLOG_FLUSH ([DLOG])

input

DLOG :	ANY_STRING	[OPTIONAL]	the name of the datalog buffer; if no name is given, then ALL the datalogs buffers are flushed

The operation only makes sense for datalogs that are configured to be persistent.
It is also actually executed only if there are samples still not flushed: finalization of samples on persistent storage is already done automatically every 5 minutes and every time the number of bufferized samples exceeds the count of 100.

DLOG_EXPORT

Starts an export on file of the samples stored in a datalog buffer.

DLOG_EXPORT (FILE, DLOG [, FROM [, TO]])

input

FILE :	ANY_STRING		path and name of the exported file
DLOG :	ANY_STRING		the name of the datalog buffer
FROM :	LDT	[OPTIONAL]	in case of limited time range, the initial timestamp of the needed interval; if given, only the samples with times from this parameter onward will be included in the export
TO :	LDT	[OPTIONAL]	in case of limited time range, the final timestamp of the needed interval; if given, only the samples with times up to this parameter will be included in the export

Multiple exports can't be executed contemporarily: using this function while another export is in progress will result in an error.

All the datalogs exports are executed asynchronously, regardless the context that started the operation (script, predefined function, grid menu, etc.); not even this function is blocking for the script, and only acts as a trigger for the operation.

Being the exports asynchronous, it is possible to have functions and script executing operations while an export is in progress.
Therefore few more functions are available to handle this asynchronicity: see **DLOG_EXPORTTERMINATE**, **DLOG_EXPORTWAIT**, **DLOG_EXPORTBUSY**.

example

```
VAR
   dlbusy : BOOL;
   dlname : STRING [32];
END_VAR;

dlname := 'DataLog1';

dlbusy := DLOG_EXPORTBUSY (dlname);     // FALSE (start in a free state)
dlbusy := DLOG_EXPORTWAIT (dlname);     // FALSE (there is nothing to wait)

DLOG_EXPORT ('d:\dl2.txt', dlname);
dlbusy := DLOG_EXPORTBUSY (dlname);     // TRUE (export just started)
// DLOG_EXPORT ('d:\dl2.txt', dlname);  // > can't request another export while the buffer is busy
// DLOG_EXPORTTERMINATE (dlname);       //   (termination might be a solution…)
dlbusy := DLOG_EXPORTWAIT (dlname);     // TRUE (an export in progress has actually been waited)
dlbusy := DLOG_EXPORTBUSY (dlname);     // FALSE (export now completed)
DLOG_EXPORT ('d:\dl2.txt', dlname);     // > it's possible to request another operation
```

DLOG_PRINT

Starts the printing of the samples stored in a datalog buffer.

DLOG_PRINT (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

Multiple prints and exports can't be executed contemporarily: using this function while another export or another print is in progress will result in an error.

Just like the **DLOG_EXPORT**, even this function works asynchronously: this function only acts as a trigger for the operation.

From every point of view, the print is considered a sort of 'export on paper', and with the export it doesn't only share the mechanics, but even the asynchronicity tools functions: see **DLOG_EXPORTTERMINATE**, **DLOG_EXPORTWAIT**, **DLOG_EXPORTBUSY**.

DLOG_EXPORTBUSY

Checks whether a datalog buffer export (or print) is currently in progress.

STATE = **DLOG_EXPORTBUSY** ([DLOG])

input			
DLOG :	ANY_STRING	[OPTIONAL]	the name of the datalog buffer if no name is given, then the check is done on ALL the datalogs

output		
STATE :	BOOL	TRUE if there is an export/print in progress; FALSE otherwise

The check is applied indiscriminately on exports and on prints in progress.

See **DLOG_EXPORT** for notes and examples about the management of the exports asynchronicity.

DLOG_EXPORTTERMINATE

Forcefully terminates the execution of an export (or print) in progress.

STATE = **DLOG_EXPORTTERMINATE** ([DLOG])

input			
DLOG :	ANY_STRING	[OPTIONAL]	the name of the datalog buffer; if no name is given, then exports and prints of ALL the datalogs are terminated

output		
STATE :	BOOL	TRUE if an export/print in progress was actually terminated; FALSE if there was nothing to terminate

The termination acts indiscriminately on exports and on prints in progress.

This function allows the script to terminate the export/print operation (for whatever reason: release the buffers for a mandatory immediate sampling, terminate all activities before a shutdown, etc.).
Nothing happens - not even errors notifications - if there is no export/print currently in progress; see the returned state.

See **DLOG_EXPORT** for notes and examples about the management of the exports asynchronicity.

DLOG_EXPORTWAIT

Waits for a datalog buffer export (or print) to complete.

STATE = **DLOG_EXPORTWAIT** ([DLOG])

input			
DLOG :	ANY_STRING	[OPTIONAL]	the name of the datalog buffer if no name is given, the system waits for exports and prints of ALL the configured datalogs

output		
STATE :	BOOL	TRUE if an export/print in progress had actually to be waited for; FALSE if there was nothing to wait for

The wait acts indiscriminately on exports and on prints in progress.

This function allows the script to wait for the whole duration of an ongoing export/print. While the script is waiting no other script can be executed, but further predefined functions could happen and the terminal would not feel blocked.
Nothing happens - not even errors notifications - if there is no export/print currently in progress; see the returned state.

See **DLOG_EXPORT** for notes and examples about the management of the exports asynchronicity.

DLOG_EXPORTCONFIG

Allows the customization of the list of fields included in exports.

DLOG_EXPORTCONFIG (DLOG, KEYS)

input

DLOG :	ANY_STRING	the name of the datalog buffer
KEYS :	ANY_STRING	a string of keys, used to specify the list of fields, along with their order and format

The configuration is buffer-specific, so each configured datalog can have its own list of fields.

The supported fields and their corresponding keys are:

date	D	date of samples acquisition
time	T	time of samples acquisition
value	V#	value of the sample; the 'V' must be followed by a sequence of numeric digits (≥ 1) used to specify the index of the field within the datalog sources collection (≥ 1); for example "V3" is used to include the value of the third source; the field output will be automatically formatted

quality	Q# | q#	quality flags of the sample;
		the 'Q' must be followed by a sequence of numeric digits (≥ 1) used to specify the index of the field within the datalog sources collection (≥ 1); for example "Q10" is used to include the quality flags of the 10th source;
		if the given key is an upper case '**Q**', then the output is formatted as a numeric value; the value is a bitstring, where each bit has the following meaning:
		bit 0 : the sample's value is valid (1) or not valid (0)
		bit 1 : the buffer automatic acquisition was enabled (1) or disabled (0)
		bit 2 : the sample is the first acquired after a start (1) or linked to previous acquisition (0)
		bit 3 : the sample is from the first element of an array source (1) or not (0)
		if the given key is a lower case '**q**' instead, then the output is formatted as a readable string made of the following segments:
		"Start" : if the sample is the first acquired after a start (bit 2 set)
		"Link" : if the sample is part of a linked acquisitions chain (bit 2 reset)
		"Invalid" : if the acquired value caused errors (bit 0 reset)
		"Disabled" : if the sample was acquired while the datalog was disabled (bit 1 reset)
		"FirstElement" : if the sample is from the first element of an array source (bit 3 set)

The following special cases are supported:
- if the given KEYS string is exactly "@Q", it means all the source qualities in the currently configured keys must become upper case "Q"s, and be displayed as numeric values;
- if the given KEYS string is exactly "@q", it means all the source qualities in the currently configured keys must become lower case "q"s, and be displayed as readable strings.

DLOG_ISENABLED

Checks whether the activity of a given datalog is currently enabled.

STATE = **DLOG_ISENABLED** (DLOG)

input

DLOG :	ANY_STRING	the name of the datalog buffer

output

STATE :	BOOL	TRUE if the datalog buffer activity is currently enabled; FALSE if it's disabled

example

```
VAR
   dlon : BOOL;
END_VAR;

dlon := DLOG_ISENABLED('DataLog1');  // initial state
DLOG_DISABLE ('DataLog1');
dlon := DLOG_ISENABLED('DataLog1');  // definitely FALSE
DLOG_ENABLE ('DataLog1');
dlon := DLOG_ISENABLED('DataLog1');  // definitely TRUE
```

DLOG_GETNUMSAMPLES

Retrieves the number of samples currently stored in a datalog buffer.

NUMBER = **DLOG_GETNUMSAMPLES** (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

output		
STATE :	UDINT	the number of stored samples

Note that all the sources of a datalog have the same number of samples.

Programmers should be careful when this counter is used as boundary in script loops: the content of the buffers could be constantly in evolution, so this counter might not be reliable if used on buffers not properly handled.

DLOG_GETSAMPLE

Retrieves the value of a sample stored in a datalog buffer.

VALUE = **DLOG_GETSAMPLE** (DLOG, SOURCEIDX, SAMPLEIDX)

input		
DLOG :	ANY_STRING	the name of the datalog buffer
SOURCEIDX :	ANY_INT	the index (base-0) of the needed source
SAMPLEIDX :	ANY_INT	the index (base-0) of the needed sample

output		
VALUE :	ANY_ELEMENTARY	the value of the SAMPLEIDXth sample currently stored in the buffer of the SOURCEIDXth source of the given datalog; the formal result type definition is enough to specify all the possible values types, but actually the only possible outcomes are: LREAL for numeric samples WSTRING for string samples

The programmer is asked to identify:
- the needed datalog, as usual with its name;
- a specific source within the datalog, identified by the index (base-0) of its position in the datalog configuration;
- a specific sample within the given datalog source buffer, identified by the index (base-0) of its position; the sample index can range between 0 and the number of samples given by **DLOG_GETNUMSAMPLES** (-1).

This function directly returns the value of the identified sample, but further information about it can be retrieved using the following dedicated variables:

DLOG_SAMPLEVALUENUM	replicates the numeric value of the sample
DLOG_SAMPLEVALUESTR	replicates the string value of the sample
DLOG_SAMPLEISSTRING	TRUE if the sample value is a string
DLOG_SAMPLETIME	gives the sample timestamp
DLOG_SAMPLEQUALITY	gives the sample quality flags

Note that the given variables are meant to retain the values coming from the last successful query invocation; in case of errors these variables values will remain unchanged.

DLOG_GETDISCARDINVALID

Gets the current behaviour model for the samples with invalid quality.

DISCARD = **DLOG_GETDISCARDINVALID** (DLOG)

input		
DLOG :	ANY_STRING	the name of the datalog buffer

output		
DISCARD :	BOOL	the currently active management model; TRUE means that the records containing samples with invalid quality (generated after communication issues) must be discarded; after a sampling operation, if the record is discarded, an 'OnSamplesError' event is generated; the datalog buffer only contains samples with good quality; FALSE means even samples with invalid quality can be stored in the buffer; when samples are added to the buffer (even samples with invalid quality) an 'OnSamplesSuccess' event is generated

DLOG_SETDISCARDINVALID

Sets a behaviour model for the management of the samples with invalid quality.

DLOG_SETDISCARDINVALID (DLOG, DISCARD)

input

DLOG :	ANY_STRING	the name of the datalog buffer
DISCARD :	BOOL	the needed management model; **TRUE** means the records containing samples with invalid quality (generated after communication issues) must be discarded; after a sampling operation, if the record is discarded, an 'OnSamplesError' event is generated; the datalog buffer only contains samples with good quality; in case a record is discarded because of quality issues, all the samples of the first valid and accepted one (after a rejection) will be marked with a discontinuity flag in their quality (**DLOGSAMPLESTART**), meant to inform about the discontinuity in the samples collection; **FALSE** means even samples with invalid quality can be stored in the buffer; when samples are added to the buffer (even samples with invalid quality) an 'OnSamplesSuccess' event is generated; this should be the preferred behaviour in case of datalogs containing multiple sources mapped on different devices, when samples with good quality coming from a device have to be retained even if in the same record exist samples with bad quality coming from others

example

```
VAR
   dlname : STRING [32] := 'DataLog1';
   dlidx, dlmax : UDINT;
   dlval : LREAL;
END_VAR;

dlmax := DLOG_GETNUMSAMPLES (dlname) - 1;

// if the source type is known…
FOR dlidx := 0 TO dlmax DO
   dlval := DLOG_GETSAMPLE (dlname, 0, dlidx);
   // LREAL variable is immediately usable if the source is known to be numeric
   // do whatever needed with <dlval> and the DLOG_SAMPLE$ vars
END_FOR;

// if the source type is NOT known…
FOR dlidx := 0 TO dlmax DO
   DLOG_GETSAMPLE (dlname, 0, dlidx);
   IF DLOG_SAMPLEISSTRING THEN
      // use value in DLOG_SAMPLEVALUESTR
      // do whatever needed with the other DLOG_SAMPLE$ vars
   ELSE
      // use value in DLOG_SAMPLEVALUENUM
      // do whatever needed with the other DLOG_SAMPLE$ vars
   END_IF;
END_FOR;
```

< VARIABLES >

The following are the variables usable to share information and directives for the datalogs management:

DLOG_SAMPLEVALUENUM
type LREAL
access R
replicates the value of the last numeric sample successfully retrieved by a call to DLOG_GETSAMPLE;
reset to 0.0 in case of string values

DLOG_SAMPLEVALUESTR
type WSTRING
access R
replicates the value of the last string sample successfully retrieved by a call to DLOG_GETSAMPLE;
reset to empty string ('') in case of numeric values

DLOG_SAMPLEISSTRING
type BOOL
access R
states whether the type of the value of the last sample successfully retrieved by a call to DLOG_GETSAMPLE is a string (TRUE) or a number (FALSE)

DLOG_SAMPLETIME
type LDT
access R
gives the acquisition timestamp of the last sample successfully retrieved by a call to DLOG_GETSAMPLE

DLOG_SAMPLEQUALITY
type BYTE
access R
gives the quality flags of the last sample successfully retrieved by a call to DLOG_GETSAMPLE;
supported flags are:

bit 0 : **1** if the sample is valid
0 if invalid (sample should be ignored)
this bitmask constant is defined: DLOGSAMPLEVALID (1)

bit 1 : **1** if the sample was acquired while the datalog was enabled
0 if it was disabled (sample should be ignored)
this bitmask constant is defined: DLOGSAMPLEENABLED (2)

bit 2 : **1** if the sample is the start of a new segment
(as after a startup or after a disable/enable)
0 if the sample can be "linked" to the previous of its segment
this bitmask constant is defined: DLOGSAMPLESTART (4)

bit 3 : **1** if the sample comes from the 1st element of an array source
0 if from other elements or from simple source tags
this bitmask constant is defined: DLOGSAMPLEFIRST (8)

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
DLOGSAMPLEVALID	1	**DLOG_GETSAMPLE**, **DLOG_SAMPLEQUALITY**
DLOGSAMPLEENABLED	2	**DLOG_GETSAMPLE**, **DLOG_SAMPLEQUALITY**
DLOGSAMPLESTART	4	**DLOG_GETSAMPLE**, **DLOG_SAMPLEQUALITY**
DLOGSAMPLEFIRST	8	**DLOG_GETSAMPLE**, **DLOG_SAMPLEQUALITY**

16. RUNTIME - USERS

Basic notes:

Most of the available users functions will rely on the concept of *"server user"*, that is the user currently considered to be "logged" in the server.

This user is the one considered to be:
- responsible for all the alarm events raised by the server
  (all the alarm events not added by clients, with or without explicit users and stations),
- responsible for the runtime events logged by the FDA auditor
  (all the runtime events not explicitly logged by clients, with or without an explicit responsible subject),
- responsible for the validation of all the users-related requests
  (originated by server-driven events).

This last characteristic in particular is relevant for the scripts' users-related functions: many operations involving users are allowed only if the responsibility lies on a user with adequate privilege levels. For example, a user with given priority levels can be created only by users with at least the same levels; or users resets can only be requested by a user at administration levels.
Therefore, for this kind of operations a responsible user must be identified:
- if the operation is requested by functions or scripts called by a client, then the responsible user is the one currently logged in that client;
- if the operation is requested by functions or scripts called due to a server event, then the *server user* is the responsible actor, and validations are based on its own levels.

By default, the *server user* is the standard "defaultuser", with the lowest priority levels.
Only clients, under appropriately validated logic, are allowed to enable a different user in the server.

To avoid the spoiling of the users system, the scripts are not allowed to:
- make authentication tests of user/password couples,
- set a new *server user*,
- browse the database of users and passwords.

The related functionalities are not implemented.

USER_ADD

Adds a new user with a given set of properties.

USER_ADD (USER, GROUP, SIGNATURE, PASSWORD, MODE [, VALIDITY [, LANGUAGE [, EMAIL [, PHONE [, RFID]]]]])

input			
USER :	ANY_STRING		name of the user
GROUP :	ANY_STRING		name of its users group
SIGNATURE :	ANY_STRING		electronic signature string
PASSWORD :	ANY_STRING		password string
MODE :	ANY_INT		the password mode; can be: **0** (USERPWDALPHA) for alphanumeric passwords **1** (USERPWDGRAPHIC) for graphic passwords
VALIDITY :	ANY_INT	[OPTIONAL]	number of days of validity of the password; the password will expire after this duration; **0** (USERPWDNOLIMIT) means the password will never expire; **-1** (USERPWDGLOBALLIMIT) means the global project duration has to be used; if missing, the default value is 365 days
LANGUAGE :	ANY_INT	[OPTIONAL]	the code of the user's default language; **0** (USERNOLANGUAGE) : there is no default language; the current language is preserved when the user logs in; **≥1** : the code is the language ID, defined as the base-1 index of the language within the project; this language will be automatically activated when the user logs in; if missing, the default value is 0 (no language)
EMAIL :	ANY_STRING	[OPTIONAL]	the e-mail address of the user; will be used by messaging (e-mail) functions for mailing lists; if missing, the default value is "" (no address available)
PHONE :	ANY_STRING	[OPTIONAL]	the telephone number of the user; will be used by messaging (SMS) functions for mailing lists; if missing, the default value is "" (no number available)
RFID :	ANY_STRING	[OPTIONAL]	the RFID alphanumeric code associated to the user's tag; if missing, or explicitly empty, the RFID is disabled for the user

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (being run in the server, the responsible user is often the current *server user*):
- the new users can't exceed the levels of the user that is trying to create them.

See the beginning of the users' chapter for notes about *server users*.

USER_REMOVE

Removes an existing user.

USER_REMOVE (USER)

input

USER :	ANY_STRING	name of the user

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (being run in the server, the responsible user is often the current *server user*):
- addressed users can't exceed the levels of the user that is trying to remove them;
- it is not possible to remove from the system the user currently active, so the USER can't be the *server user*.
See the beginning of the users' chapter for notes about *server users*.

USER_SETPASSWORD

Changes the password of an existing user.

USER_SETPASSWORD (USER, PASSWORD, MODE)

input		
USER :	ANY_STRING	name of the user
PASSWORD :	ANY_STRING	password string
MODE :	ANY_INT	the password mode; can be: **0** (USERPWDALPHA) for alphanumeric passwords **1** (USERPWDGRAPHIC) for graphic passwords

In case of graphic passwords, the given password string MUST be a WSTRING.
Each character of the string describes a node of the graphic password.
The 16-bits-characters must have the following layout:
- bits 0..3 = Y coordinate of the node (base-0)
- bits 4..7 = X coordinate of the node (base-0)
- bits 8..15 = 0xFF

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (being run in the server, the responsible user is often the current *server user*):
- referenced users can't exceed the levels of the user that is trying to update them;
- passwords of users imported from an Active Directory can only be changed by the user himself.

See the beginning of the users' chapter for notes about *server users*.

USER_SETVALIDITY

Changes the number of days of validity of the password of an existing user.

USER_SETVALIDITY (USER, VALIDITY)

input

USER :	ANY_STRING	name of the user
VALIDITY :	ANY_INT	number of days of validity of the password **0** (USERPWDNOLIMIT) means the password will never expire; **-1** (USERPWDGLOBALLIMIT) means the global project duration has to be used

After the given time (if given) the password of the user will expire and will no longer be usable.
Passwords' validity counters reset when:
- the user is created,
- a new password is given for the user,
- a new validity time is given for the password.

USER_SETGROUP

Assigns a new group to an existing user.

USER_SETGROUP (USER, GROUP)

input		
USER :	ANY_STRING	name of the user
GROUP :	ANY_STRING	name of its users group

The group will define the privilege levels of the user.

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (being run in the server, the responsible user is often the current *server user*):
- referenced users can't exceed the levels of the user that is trying to update them;
- referenced groups can't exceed the levels of the user that is trying to assign them.

See the beginning of the users' chapter for notes about *server users*.

USER_SETSIGNATURE

Changes the electronic signature string of an existing user.

USER_SETSIGNATURE (USER, SIGNATURE)

input		
USER :	ANY_STRING	name of the user
SIGNATURE :	ANY_STRING	the electronic signature of the user

The signature is used when needed in validation and export operations.

USER_SETRFID

Changes the RFID code string of an existing user.

USER_SETRFID (USER, RFID)

input		
USER :	ANY_STRING	name of the user
RFID :	ANY_STRING	the RFID code of the user

The RFID code is used for user authentication in case of logins with RFID tags.

USER_SETLANGUAGE

Changes the default language of an existing user.

USER_SETLANGUAGE (USER, LANGUAGE)

input

USER :	ANY_STRING	name of the user
LANGUAGE :	ANY_INT	the code of the user's default language; **0** (USERNOLANGUAGE) : there is no default language; the current language is preserved when the user logs in; **≥1** : the code is a language ID, defined as the base-1 index of the language within the project; this language will be automatically activated when the user logs in

If defined, the given language will be automatically activated every time the user logs in.

USER_SETEMAIL

Changes the e-mail address of an existing user.

USER_SETEMAIL (USER, EMAIL)

input		
USER :	ANY_STRING	name of the user
EMAIL :	ANY_STRING	the e-mail address of the user

The address will be used by messaging (e-mail) functions for mailing lists.

USER_SETTELNUMBER

Changes the telephone number of an existing user.

USER_SETTELNUMBER (USER, PHONE)

input		
USER :	ANY_STRING	name of the user
PHONE :	ANY_STRING	the telephone number of the user

The number will be used by messaging (SMS) functions for mailing lists.

USER_LOCK

Locks an existing user, making it unable to login.

USER_LOCK (USER)

input		
USER :	ANY_STRING	name of the user

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (the responsible user could be either the current *server user* or the user logged in the calling client):
- referenced users must have lower levels than the user that is trying to lock them.

Note that users groups can be configured to reject lock requests.
Also groups can be configured to reject UNlock requests; in this last case, lock requests become permanent, and the affected users will be marked consequently. A permanent lock state can also be forced by an explicit call to a **USER_PERMANENTLOCK** function.

See the beginning of the users' chapter for notes about *server users*.

USER_UNLOCK

Unlocks an existing user, making it able to log in again.

USER_UNLOCK (USER)

input		
USER :	ANY_STRING	name of the user

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (the responsible user could be either the current *server user* or the user logged in the calling client):
- referenced users must have lower levels than the user that is trying to unlock them.

Note that users groups can be configured to reject unlock requests.
Also note that permanently locked users can't be unlocked.

See the beginning of the users' chapter for notes about *server users*.

USER_PERMANENTLOCK

Permanently locks an existing user, making it unable to login, and unable to be unlocked.

USER_PERMANENTLOCK (USER)

input

USER :	ANY_STRING	name of the user

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (the responsible user could be either the current *server user* or the user logged in the calling client):
- referenced users must have lower levels than the user that is trying to lock them.

Note that users groups can be configured to reject lock requests.

See the beginning of the users' chapter for notes about *server users*.

USER_JOINLIST

Adds an existing user to a given mailing list.

USER_JOINLIST (USER, LIST, RECIPIENT)

input

USER :	ANY_STRING	name of the user
LIST :	ANY_STRING	name of the mailing list
RECIPIENT :	ANY_INT	recipient type when an e-mail is sent to this user as part of the given mailing list; possible codes are: **0** (USERTYPENORMAL) **1** (USERTYPECOPY) **2** (USERTYPEHIDDEN)

USER_LEAVELIST

Removes a user from a given mailing list.

USER_LEAVELIST (USER, LIST)

input		
USER :	ANY_STRING	name of the user
LIST :	ANY_STRING	name of the mailing list

USER_RESETLISTS

Removes a user from all of its mailing lists.

USER_RESETLISTS (USER)

input

USER :	ANY_STRING	name of the user

USER_GETCURRENTNAME

Retrieves the name of the current *server user*.

NAME = **USER_GETCURRENTNAME** ()

output

NAME :	WSTRING	name of the current *server user*

See the beginning of the users' chapter for notes about *server users*.

USER_GETCURRENTGROUP

Retrieves the group of the current *server user*.

GROUP = **USER_GETCURRENTGROUP** ()

output		
GROUP :	WSTRING	name of the group of the current *server user*

See the beginning of the users' chapter for notes about *server users*.

USER_GETCURRENTSHOW

Retrieves the visualization privilege level of the current *server user*.
This function can be used only in case of **level-users** (an error would be raised otherwise).

```
LEVEL = USER_GETCURRENTSHOW ()
```

output		
LEVEL :	UDINT	visualization privilege level of the current *server user*

See the beginning of the users' chapter for notes about *server users*.

USER_GETCURRENTUSE

Retrieves the interaction privilege level of the current *server user*.
This function can be used only in case of **level-users** (an error would be raised otherwise).

```
LEVEL = USER_GETCURRENTUSE ()
```

output		
LEVEL :	UDINT	interaction privilege level of the current *server user*

See the beginning of the users' chapter for notes about *server users*.

USER_GETGROUP

Retrieves the name of the group of a given user.

GROUP = **USER_GETGROUP** (USER)

input		
USER :	ANY_STRING	name of the user

output		
GROUP :	WSTRING	name of the group of the given user

USER_GETLEVELSHOW

Retrieves the visualization privilege level of a given user.
This function can be used only in case of **level-users** (an error would be raised otherwise).

LEVEL = **USER_GETLEVELSHOW** (USER)

input		
USER :	ANY_STRING	name of the user

output		
LEVEL :	UDINT	visualization privilege level of the given user

USER_GETLEVELUSE

Retrieves the interaction privilege level of a given user.
This function can be used only in case of **level-users** (an error would be raised otherwise).

LEVEL = **USER_GETLEVELUSE** (USER)

input		
USER :	ANY_STRING	name of the user

output		
LEVEL :	UDINT	interaction privilege level of the given user

USER_GETLANGUAGE

Retrieves the default language of a given user.

LANGUAGE = **USER_GETLANGUAGE** (USER)

input		
USER :	ANY_STRING	name of the user

output		
LANGUAGE :	UDINT	ID of the default language of the given user; possible result values are: **0** (USERNOLANGUAGE) : the user has no default language; **≥1** : the code is a language ID, defined as the base-1 index of the language within the project

USER_GETEMAIL

Retrieves the e-mail address of a given user.

EMAIL = **USER_GETEMAIL** (USER)

input		
USER :	ANY_STRING	name of the user

output		
EMAIL :	WSTRING	e-mail address of the given user; could be an empty string if no address has been provided

USER_GETTELNUMBER

Retrieves the telephone number of a given user.

PHONE = **USER_GETTELNUMBER** (USER)

input		
USER :	ANY_STRING	name of the user

output		
PHONE :	STRING	telephone number of the given user could be an empty string if no number has been provided

USER_GETVALIDITY

Retrieves the number of days of validity of the password of a given user.

VALIDITY = **USER_GETVALIDITY** (USER)

input		
USER :	ANY_STRING	name of the user

output		
VALIDITY :	UDINT	number of days of validity of the password of the given user **0** (USERPWDNOLIMIT) means the password is set not to expire

USER_GETCREATION

Retrieves the creation date of the password of a given user.

DATE = **USER_GETCREATION** (USER)

input		
USER :	ANY_STRING	name of the user

output		
DATE :	DATE	creation date of the password

USER_GETEXPIRATION

Retrieves the expiration date of the password of a given user.

DATE = **USER_GETEXPIRATION** (USER)

input		
USER :	ANY_STRING	name of the user

output		
DATE :	DATE	expiration date of the password

USER_ISLOCKED

Checks whether a given user is currently locked.

STATE = **USER_ISLOCKED** (USER)

input		
USER :	ANY_STRING	name of the user

output		
STATE :	UDINT	the current lock state of the user; can be: **0** (USERUNLOCKED) means the user is not locked **1** (USERLOCKED) means the user is locked **2** (USERPERMANENTLOCK) means the user is permanently locked

USER_ISIMPORTED

Checks whether a given user was imported from an Active Directory.

STATE = **USER_ISIMPORTED** (USER)

input		
USER :	ANY_STRING	name of the user

output		
STATE :	BOOL	TRUE if the given user was imported from an Active Directory; FALSE otherwise

USER_HASRFID

Checks whether a given user supports RFID authentication.

STATE = **USER_HASRFID** (USER)

input		
USER :	ANY_STRING	name of the user

output		
STATE :	BOOL	TRUE if the given user supports RFID authentication; FALSE otherwise

USER_GROUPGETNAME

Retrieves the name of a group with a known ID.

GROUP = **USER_GROUPGETNAME** (ID)

input		
ID :	UDINT	ID of the users group; the ID is defined as the index (base-0) of the group within the project

output		
GROUP :	WSTRING	name of the users group

Scripts don't normally reference users groups through IDs; their usage though allows the programmer to browse the names of the existing groups.

USER_GROUPGETID

Retrieves the ID of a group with a known name.

ID = **USER_GROUPGETID** (GROUP)

input		
GROUP :	ANY_STRING	name of the users group

output		
ID :	UDINT	ID of the users group; the ID is defined as the index (base-0) of the group within the project

Scripts don't normally reference users groups through IDs; their usage though allows the programmer to browse the names of the existing groups.

USER_GROUPLEVELSHOW

Retrieves the visualization privilege level of the users of a given group.
This function can be used only in case of **level-users** (an error would be raised otherwise).

LEVEL = **USER_GROUPLEVELSHOW** (GROUP)

input		
GROUP :	ANY_STRING	name of the users group

output		
LEVEL :	UDINT	visualization privilege level of the users of the given group

USER_GROUPLEVELUSE

Retrieves the interaction privilege level of the users of a given group.
This function can be used only in case of **level-users** (an error would be raised otherwise).

LEVEL = **USER_GROUPLEVELUSE** (GROUP)

input		
GROUP :	ANY_STRING	name of the users group

output		
LEVEL :	UDINT	interaction privilege level of the users of the given group

USER_FLUSH

Flushes on persistent storage the records related to users' events (the records could still be held in system buffers or in files caches).

USER_FLUSH ()

This function is meant to be used in cases where the FDA auditor is not enabled, and the only logged runtime events are those coming from the users.
In case the FDA functionalities are enabled, being the users' events a subset of those logged by the FDA itself, this function is exactly the same as the **AUDIT_FLUSH**.

USER_EXPORT

Exports on file the records (all or in part) logged due to users' actions and changes.

USER_EXPORT (FILE, MODE, [, FROM [, TO]])

input

FILE :	ANY_STRING		path and name of the exported file
MODE :	ANY_INT		type of the exported file; can be one of the following: **0** (**FILECSV**) the file is exported in CSV mode **1** (**FILEPDF**) the file is exported in PDF mode
FROM :	ANY_DATE	[OPTIONAL]	timestamp of the oldest exported record this parameter can actually be of the following types only: DT, LDT, DATE
TO :	ANY_DATE	[OPTIONAL]	timestamp of the most recent exported record this parameter can actually be of the following types only: DT, LDT, DATE

The 3rd and 4th parameters are optional: they are meant to allow the export of specific ranges of records, starting from, and terminating to, given time markers.
If both parameters are missing, the export method will export all the logged records;
if at least the 3rd parameter exists, the export method will not export records older than the given time;
if the 4th parameter exists too, the export method will not export records newer than the given time; otherwise it will go on up to the end of the log.

FILE, MODE	export the whole file
FILE, MODE, **FROM**	export records with at least the FROM timestamp
FILE, MODE, **FROM**, **TO**	export records with timestamps between FROM and TO

Note that in case the FDA auditor functionalities are enabled, the users' events records are essentially a subset of the records logged by the FDA itself. Whereas the **AUDIT_EXPORT** exports everything though, the records exported by this function are still limited to those strictly related to the users.

example

```
USER_EXPORT ('/home/esa/test1.txt', FILECSV);
USER_EXPORT ('/home/esa/test2.txt', FILECSV, MAKELDT(2019,1,1,0,0,0,0));
USER_EXPORT ('/home/esa/test3.txt', FILEPDF, DT#2019-1-1-0:0:0, DT#2019-2-1-0:0:0);
```

USER_PRINT

Prints the records (all or in part) logged due to users' actions and changes.

USER_PRINT ([FROM [, TO]])

input

FROM :	ANY_DATE	[OPTIONAL]	timestamp of the oldest printed record this parameter can actually be of the following types only: DT, LDT, DATE
TO :	ANY_DATE	[OPTIONAL]	timestamp of the most recent printed record this parameter can actually be of the following types only: DT, LDT, DATE

The 2nd and 3rd parameters are optional: they are meant to allow the export of specific ranges of records, starting from, and terminating to, given time markers.
See **USER_EXPORT** for notes and *examples* regarding the usage of the time range parameters.

Note that in case the FDA auditor functionalities are enabled, the users' events records are essentially a subset of the records logged by the FDA itself. Whereas the **AUDIT_PRINT** prints everything though, the records printed by this function are still limited to those strictly related to the users.

USER_RESET

Resets the users, with all their passwords and properties, to the state originally defined in the project.

USER_RESET ()

All the changes made at runtime will be lost.

This operation is subject to validation, based on the privilege levels of the user responsible for the script execution (being run in the server, the responsible user is often the current *server user*):
- this operation is reserved to users with recognized administrative levels: this means only users with both visualization and interaction privilege levels set to 1 (maximum priority).
See the beginning of the users' chapter for notes about *server users*.

USER_IMPORTNETWORK

Imports the users list from the Active Directory of the network specified in the project.

USER_IMPORTNETWORK ([USER, PASSWORD [, NETWORK, PATH, FILTER]])

input

USER :	ANY_STRING	[OPTIONAL]	user-name for network authentication
PASSWORD :	ANY_STRING	[OPTIONAL]	user password for network authentication
NETWORK :	ANY_STRING	[OPTIONAL]	name of the network server; can also be the server IP address in string form
PATH :	ANY_STRING	[OPTIONAL]	path of an Active Directory branch; only users from the given AD part will be imported
FILTER :	ANY_STRING	[OPTIONAL]	filter for Active Directory records recognition; only users matching the given filter will be imported

Programmers are allowed to invoke this method with 3 parameters configurations only:
- giving no parameter at all;
- giving only the first 2 parameters;
- giving all the listed parameters.

The 1st and 2nd modes are the recommended ones; the 3rd might not be ideal, depending on the programmer expectations. In details:
1) if no parameter is given, then the method will automatically use the properties given in the project Active Directory configuration; this will ensure an optimal management of the Active Directory;
2) if only the first two parameters are given, then the programmer is still using the pre-configured Active Directory, but intends to access it with a different/one-time authentication; the optimal management of the Active Directory is still granted;
3) if all the parameters are given, then the programmer is referencing an Active Directory different from the pre-configured one; in this case the users are imported from the specified network, but afterward they will only be able to authenticate themselves in offline mode: automatic runtime operations like authentications or updates of password expiration dates are always directed to the pre-configured Active Directory, so users imported from alternate networks will not be able to interact with their network server, and will be forced to work as simple runtime offline users (whereas users imported from the pre-configured network will be authenticated by the Active Directory server itself).

example

```
USER_IMPORTNETWORK ();
USER_IMPORTNETWORK ('MyName', 'MyPwd', 'esahmi.lan', 'DC=esahmi,DC=lan', '(objectCategory=person)');
```

USER_EXPORTGROUPMATRIX

Export the whole group permissions matrix.
This function can be used only in case of **matrix-users** (an error would be raised otherwise).

USER_EXPORTGROUPMATRIX (FILE)

input		
FILE :	ANY_STRING	path and name of exported file

USER_IMPORTGROUPMATRIX

Import a whole group permissions matrix.
This function can be used only in case of **matrix-users** (an error would be raised otherwise).

AUTHORIZATIONS = **USER_IMPORTGROUPMATRIX** (FILE)

input		
FILE :	ANY_STRING	path and name of imported file

output		
AUTHORIZATIONS :	UDINT	number of imported authorizations (number of columns in the matrix); note that the number of rows is fixed since it must exactly match the number of groups configured in the project

USER_EXPORTGEOMATRIX

Export the whole geographic permissions matrix.
This function can be used only in case of **matrix-users** (an error would be raised otherwise).

USER_EXPORTGEOMATRIX (FILE)

input		
FILE :	ANY_STRING	path and name of exported file

USER_IMPORTGEOMATRIX

Import a whole geographic permissions matrix.
This function can be used only in case of **matrix-users** (an error would be raised otherwise).

AUTHORIZATIONS = **USER_IMPORTGEOMATRIX** (FILE)

input		
FILE :	ANY_STRING	path and name of imported file

output		
AUTHORIZATIONS :	UDINT	number of imported authorizations (number of columns in the matrix); note that the number of rows is fixed since it must exactly match the number of client machines configured in the project

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
USERPWDALPHA	0	**USER_ADD**, **USER_SETPASSWORD**
USERPWDGRAPHIC	1	**USER_ADD**, **USER_SETPASSWORD**
USERPWDNOLIMIT	0	**USER_ADD**, **USER_SETVALIDITY**, **USER_GETVALIDITY**
USERPWDGLOBALLIMIT	0xFFFFFFFF (-1)	**USER_ADD**, **USER_SETVALIDITY**
USERNOLANGUAGE	0	**USER_ADD**, **USER_SETLANGUAGE**, **USER_GETLANGUAGE**
USERTYPENORMAL	0	**USER_JOINLIST**
USERTYPECOPY	1	**USER_JOINLIST**
USERTYPEHIDDEN	2	**USER_JOINLIST**
USERUNLOCKED	0	**USER_ISLOCKED**
USERLOCKED	1	**USER_ISLOCKED**
USERPERMANENTLOCK	2	**USER_ISLOCKED**

17. RUNTIME - PIPELINES

PIPELINE_ENABLE

Enables the automatic activity of a pipeline.

PIPELINE_ENABLE (PIPELINE)

input

PIPELINE :	ANY_STRING	name of the needed pipeline

Note that it is not possible to enable or disable pipelines with manual (COMMAND) trigger, since these pipelines have no automatic activity; the operation is only allowed with pipelines with execution mode VARIATION, CONTINUOUS, TAGTRIGGER.
Also it is not possible to enable or disable pipelines with enabled mode ALWAYSON, since they are explicitly configured to forbid these changes.

PIPELINE_DISABLE

Disables the automatic activity of a pipeline.

PIPELINE_DISABLE (PIPELINE)

input

PIPELINE :	ANY_STRING	name of the needed pipeline

Note that it is not possible to enable or disable pipelines with manual (COMMAND) trigger, since these pipelines have no automatic activity; the operation is only allowed with pipelines with execution mode VARIATION, CONTINUOUS, TAGTRIGGER.
Also it is not possible to enable or disable pipelines with enabled mode ALWAYSON, since they are explicitly configured to forbid these changes.

PIPELINE_WRITE

Forces the execution of a single source-destination copy of a pipeline.

PIPELINE_WRITE (PIPELINE)

input		
PIPELINE :	ANY_STRING	name of the needed pipeline

PIPELINE_ISENABLED

Checks whether a given pipeline is currently enabled or disabled.

STATE = **PIPELINE_ISENABLED** (PIPELINE)

input		
PIPELINE :	ANY_STRING	name of the needed pipeline

output		
STATE :	BOOL	TRUE if the pipeline is currently enabled; FALSE otherwise

PIPELINE_GETID

Retrieves the ID of a pipeline with a given name.

ID = **PIPELINE_GETID** (PIPELINE)

input		
PIPELINE :	ANY_STRING	name of the needed pipeline

output		
ID :	UDINT	the ID of the specified pipeline; IDs are defined as numeric indexes (base-1) arranged in a continuous sequence

The pipelines IDs are not actually needed by scripts programmer, since all the relevant methods expect the usage of names instead.
This method is currently mostly intended for debugging purposes; in combination with **PIPELINE_GETNAME** and **PIPELINE_GETNUMBER**, it allows to retrieve the complete list of the configured pipelines.

PIPELINE_GETNAME

Retrieves the name of a pipeline with a given ID.

PIPELINE = **PIPELINE_GETNAME** (ID)

input		
ID :	UDINT	the ID of the needed pipeline; IDs are defined as numeric indexes (base-1) arranged in a continuous sequence; the value can range between 1 and **PIPELINE_GETNUMBER**

output		
PIPELINE :	WSTRING	name of the specified pipeline

PIPELINE_GETNUMBER

Retrieves the number of pipelines configured in the current project.

NUMBER = **PIPELINE_GETNUMBER** ()

output		
NUMBER :	UDINT	number of configured pipelines

18. RUNTIME - FDA

AUDIT_ENABLE

Enables the FDA auditor logging activity.

AUDIT_ENABLE ()

AUDIT_DISABLE

Disables the FDA auditor logging activity.

AUDIT_DISABLE ()

This method can be used only if the FDA system has been configured to actually support it: the FDA system can be configured to disallow these requests; if this is the case, the method will fail.

example

```
VAR
   enabled : BOOL;
END_VAR;

enabled := AUDIT_ISENABLED ();     // initial enabled state

ST_OPTION HANDLE_ERRORS;           // enable blocking errors management
   ERRORRESET ();
   AUDIT_DISABLE ();
ST_OPTION BLOCKING_ERRORS;

IF ERRNO = 0 THEN
   // the DISABLE request succeeded
ELSE
   // the DISABLE request failed; might want to check the FDA configuration
END_IF;

enabled := AUDIT_ISENABLED ();     // final enabled state
```

AUDIT_FLUSH

Flushes on persistent storage the records logged by the FDA auditor (the records could still be held in system buffers or in files caches).

AUDIT_FLUSH ()

AUDIT_EXPORT

Exports on file the records (all or in part) logged by the FDA auditor.

AUDIT_EXPORT (FILE, MODE, SIGNATURE, [, FROM [, TO]])

input

FILE :	ANY_STRING		path and name of the exported file
MODE :	ANY_INT		type of the exported file; can be one of the following: **0** (**FILECSV**) the file is exported in CSV mode **1** (**FILEPDF**) the file is exported in PDF mode
SIGNATURE :	ANY_STRING		signature entered by the user requesting the export; if no signature is used, then an empty string has to be explicitly given
FROM :	ANY_DATE	[OPTIONAL]	timestamp of the oldest exported record this parameter can actually be of the following types only: DT, LDT, DATE
TO :	ANY_DATE	[OPTIONAL]	timestamp of the most recent exported record this parameter can actually be of the following types only: DT, LDT, DATE

The 4th and 5th parameters are optional: they are meant to allow the export of specific ranges of records, starting from, and terminating to, given time markers.
If both parameters are missing, the export method will export all the records in the log;
if at least the 4th parameter exists, the export method will not export records older than the given time;
if the 5th parameter exists too, the export method will not export records newer than the given time; otherwise it will go on up to the end of the log.

FILE, …	export the whole file
FILE, …, **FROM**	export records with at least the FROM timestamp
FILE, …, **FROM**, **TO**	export records with timestamps between FROM and TO

example

```
AUDIT_EXPORT ('/home/esa/test1.txt', FILECSV, '');
AUDIT_EXPORT ('/home/esa/test2.txt', FILECSV, '', MAKELDT(2019,1,1,0,0,0,0));
AUDIT_EXPORT ('/home/esa/test3.txt', FILECSV, '', DT#2019-1-1-0:0:0, DT#2019-2-1-0:0:0);
```

AUDIT_RESET

Operates a full reset of the FDA auditor logs.

AUDIT_RESET (FILE, MODE, SIGNATURE)

input		
FILE :	ANY_STRING	path and name of the exported file
MODE :	ANY_INT	type of the exported file; can be one of the following: **0** (FILECSV) the file is exported in CSV mode **1** (FILEPDF) the file is exported in PDF mode
SIGNATURE :	ANY_STRING	signature entered by the user requesting the reset; if no signature is used, then an empty string has to be explicitly given

The reset includes:
- a forced full export of the currently logged records;
- a cleanup of the whole log;
- a reset of a possible blocking error keeping the auditor inactive.

Note that only users with maximum (administration) rights are allowed to invoke the reset directive.

AUDIT_PRINT

Prints the records (all or in part) logged by the FDA auditor.

AUDIT_PRINT ([FROM [, TO]])

input			
FROM :	ANY_DATE	[OPTIONAL]	timestamp of the oldest printed record this parameter can actually be of the following types only: DT, LDT, DATE
TO :	ANY_DATE	[OPTIONAL]	timestamp of the most recent printed record this parameter can actually be of the following types only: DT, LDT, DATE

The 2nd and 3rd parameters are optional: they are meant to allow the export of specific ranges of records, starting from, and terminating to, given time markers.
See **AUDIT_EXPORT** for notes and *examples* regarding the usage of the time range parameters.

AUDIT_ISENABLED

Checks whether the FDA auditor is currently enabled or disabled.

STATE = **AUDIT_ISENABLED** ()

output

STATE :	BOOL	TRUE if the FDA auditor is currently enabled; FALSE otherwise

AUDIT_GETERROR

Retrieves the code of the error that is keeping the FDA auditor inactive.

```
ERROR = AUDIT_GETERROR ()
```

output		
ERROR :	ULINT	code of the blocking error

If there is no error, this function returns 0; otherwise a non-0 error code.
This kind of blocking errors can happen when the FDA system is configured to be mandatory for the system.
When blocking errors happen, they will persist through system restarts; the only way to remove them is a complete FDA system reset (see **AUDIT_RESET**).

AUDIT_GETNUMBER

Retrieves the number of records currently logged by the FDA auditor.

NUMBER = **AUDIT_GETNUMBER** ()

output

NUMBER :	UDINT	number of logged records

AUDIT_READ

Reads a record from the FDA auditor logs.

ID = **AUDIT_READ** (INDEX)

input

INDEX :	ANY_INT	index (base-0) of the needed record can be any value between 0 and the number of logged records (-1); the number of records can be retrieved with a call to AUDIT_GETNUMBER

output

ID :	UDINT	the ID of the retrieved record; in case of successful readings, this could be any positive value; in case of failure, the returned value is AUDITNOID (0xFFFFFFFF)

After a record has been successfully read, the following variables can be used to retrieve its data:

AUDIT_READMODULE	name of the related runtime module
AUDIT_READACTION	name of the related runtime action
AUDIT_READTIME	timestamp of the record (time of the logged action)
AUDIT_READUSER	name of the responsible user
AUDIT_READCLIENT	name of the responsible client (matrix users only)
AUDIT_READOBJECT	description of the related runtime object
AUDIT_READCOMMENT	comment associated to the event
AUDIT_READREASON	operation reason entered by user at runtime
AUDIT_READSIGNATURE	electronic signature of involved user
AUDIT_READTAGNAME	name of the tag involved in the event
AUDIT_READTAGADD	symbolic address of the tag involved in the event
AUDIT_READOLDVALUE	old tag value (in case of edited tag event)
AUDIT_READNEWVALUE	new tag value (in case of edited tag event)

Note that the given variables are meant to retain the values coming from the last successful read function invocation; in case of errors (AUDITNOID) returned by this AUDIT_READ, these variables values will remain unchanged.

example

```
VAR
   recnum : UDINT;
   recidx : UDINT;
   recid  : UDINT;
   record : WSTRING [256];
END_VAR;

recnum := AUDIT_GETNUMBER ();
FOR recidx := 0 TO recnum-1 DO
   recid := AUDIT_READ (recidx);
   IF (recid <> AUDITNOID) THEN
      record := ANY_TO_STRING(AUDIT_READTIME) + ' -  M : ' + AUDIT_READMODULE
                                              + ' ,  A : ' + AUDIT_READACTION;
      // ... do whatever needed with the retrieved information ...

   END_IF;
END_FOR;
```

< VARIABLES >

The following are the variables usable to share information and directives for the auditor management:

AUDIT_READMODULE
type STRING
access R
gives the name of the runtime module related to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READACTION
type STRING
access R
gives the name of the runtime event/action related to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READTIME
type LDT
access R
gives the timestamp of the logged record returned by the last successful call to AUDIT_READ

AUDIT_READUSER
type STRING
access R
gives the name of the responsible user related to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READCLIENT
type STRING
access R
gives the name of the responsible client related to the logged record returned by the last successful call to AUDIT_READ (meaningful in case of matrix users only)

AUDIT_READOBJECT
type STRING
access R
gives a description of the runtime object related to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READCOMMENT
type STRING
access R
gives the embedded comment appended to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READREASON
type STRING
access R
gives the reason entered at runtime by the user, appended to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READSIGNATURE
type STRING
access R
gives the electronic signature entered by the involved user (if needed by user and required by specific event, only loggable by UI-clients events) appended to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READTAGNAME
type STRING
access R

gives the name of the tag appended to the logged record returned by the last successful call to AUDIT_READ (meaningful in case of tag-events only)

AUDIT_READTAGADD
type STRING
access R
gives the symbolic address of the tag appended to the logged record returned by the last successful call to AUDIT_READ (meaningful in case of tag-events only, and only in case of tags with symbolic address)

AUDIT_READOLDVALUE
type STRING
access R
gives the old tag value (in case of tag editing event, only loggable by UI-clients) appended to the logged record returned by the last successful call to AUDIT_READ

AUDIT_READNEWVALUE
type STRING
access R
gives the new tag value (in case of tag editing event, only loggable by UI-clients) appended to the logged record returned by the last successful call to AUDIT_READ

< CONSTANTS >

The following symbolics can be used as numeric constants.
The symbolics could be used anywhere in the scripts, regardless the context, and are treated exactly as if the corresponding numeric value were written in their place.
They are intended, though, to be used as self-explanatory values while passing arguments to the relevant methods:

symbolic	value	relevant methods
AUDITNOID	0xFFFFFFFF (-1)	**AUDIT_READ**

Connect ideas.

shape solutions.