

---

GRID AUTOMATION PRODUCTS

# **MicroSCADA X SYS600 10.2**

## Programming Language SCIL







Document ID: 1MRK 511 479-UEN  
Issued: March 2021  
Revision: A  
Product version: 10.2

© 2021 Hitachi Power Grids. All rights reserved.



## Table of contents

<b>Section 1</b>	<b>Copyrights.....</b>	<b>21</b>
<b>Section 2</b>	<b>About this manual.....</b>	<b>23</b>
2.1	Use of symbols.....	23
2.2	Intended audience.....	23
2.3	Related documents.....	23
2.4	Document conventions.....	23
2.5	Document revisions.....	24
<b>Section 3</b>	<b>Introduction.....</b>	<b>25</b>
3.1	Presentation of SCIL.....	25
3.1.1	What is SCIL.....	25
3.1.2	Application engineering.....	25
3.1.3	What can be done with SCIL.....	26
3.2	SCIL programs.....	27
3.2.1	Composition.....	27
3.2.2	Occurrence .....	28
3.2.3	Activation of SCIL programs.....	28
3.2.4	Execution modes.....	28
3.2.5	Example.....	28
3.3	SCIL statements.....	29
3.3.1	Components.....	29
3.3.2	Statement format.....	29
3.4	Organization of this manual.....	30
<b>Section 4</b>	<b>Programming in SCIL.....</b>	<b>31</b>
4.1	SCIL programming environment.....	31
4.1.1	General.....	31
4.1.2	SCIL program editor.....	31
4.1.3	Picture programs.....	31
4.1.4	Window definitions.....	32
4.1.5	Methods.....	33
4.1.6	Command procedures.....	33
4.1.7	Data objects.....	34
4.1.8	Time channels.....	35
4.1.9	Using SCIL in external applications.....	36
4.1.10	SCIL test tool.....	37
4.2	SCIL programming rules.....	37
4.2.1	Program structure.....	37
4.2.2	Examples.....	37
4.2.3	SCIL characters.....	38
4.2.4	SCIL names.....	39

<b>Section 5</b>	<b>Data types.....</b>	<b>41</b>
5.1	General.....	41
5.1.1	Data types.....	41
5.1.2	Reliability of data.....	41
5.2	Integer.....	41
5.2.1	Description.....	41
5.2.2	Example.....	42
5.3	Real.....	42
5.3.1	Description.....	42
5.3.2	Example.....	43
5.4	Boolean.....	43
5.4.1	Description.....	43
5.4.2	Examples.....	43
5.5	Time.....	43
5.6	Text.....	44
5.6.1	Description.....	44
5.6.2	Examples.....	44
5.7	Bit string.....	44
5.7.1	Description.....	44
5.7.2	Example.....	44
5.8	Byte string .....	45
5.8.1	Description.....	45
5.9	Vector.....	45
5.9.1	Description.....	45
5.9.2	Vector aggregate.....	45
5.9.3	Example.....	45
5.10	List.....	46
5.10.1	Description.....	46
5.10.2	List aggregate.....	46
5.10.3	Example.....	46
5.11	Accessing components of structured data.....	46
<b>Section 6</b>	<b>Objects and files.....</b>	<b>49</b>
6.1	General.....	49
6.1.1	Object categories.....	49
6.1.2	Attributes and methods.....	50
6.1.3	Handling objects in SCIL.....	50
6.2	System objects.....	50
6.2.1	General.....	50
6.2.1.1	Overview.....	50
6.2.1.2	System object notation.....	51
6.2.2	Base system objects (B).....	51
6.2.2.1	Description.....	51
6.2.2.2	Some attributes.....	52
6.2.2.3	Examples.....	52
6.2.3	Communication system objects.....	52

6.2.3.1	Description.....	52
6.2.3.2	Some attributes.....	53
6.2.3.3	Examples.....	53
6.3	Application objects.....	53
6.3.1	General.....	53
6.3.1.1	Object types.....	53
6.3.1.2	Application object notation.....	54
6.3.2	Process objects (P).....	55
6.3.2.1	Description.....	55
6.3.2.2	Some process object attributes.....	55
6.3.2.3	Groups and indices.....	56
6.3.2.4	Examples.....	56
6.3.3	Event handling objects (H).....	57
6.3.3.1	Description.....	57
6.3.3.2	Some event handling attributes.....	57
6.3.4	Scales (X).....	57
6.3.4.1	Description.....	57
6.3.4.2	Some scale attributes.....	57
6.3.5	Data objects (D).....	58
6.3.5.1	Description.....	58
6.3.5.2	Some data object attributes.....	58
6.3.5.3	Examples.....	58
6.3.6	Command procedures (C).....	59
6.3.6.1	Description.....	59
6.3.6.2	Some command procedure attributes.....	59
6.3.6.3	Examples .....	60
6.3.7	Time channels (T).....	60
6.3.7.1	Description.....	60
6.3.7.2	Some time channel attributes.....	60
6.3.8	Event channels (A).....	61
6.3.8.1	Description.....	61
6.3.8.2	Some event channel attributes.....	61
6.3.9	Logging profiles (G).....	61
6.3.9.1	Description.....	61
6.3.10	Event objects (E).....	62
6.3.10.1	Description.....	62
6.3.10.2	Example.....	62
6.3.11	Variable objects (V).....	63
6.3.11.1	Description.....	63
6.3.11.2	Examples.....	63
6.4	User interface objects.....	64
6.4.1	Visual SCIL objects.....	64
6.4.1.1	General.....	64
6.4.1.2	Dialog systems.....	64
6.4.1.3	Attributes and methods.....	65
6.4.1.4	Visual SCIL object references.....	65

6.4.1.5	Attribute references.....	66
6.4.1.6	Method calls.....	66
6.4.2	Pictures.....	67
6.4.2.1	General.....	67
6.4.2.2	Picture hierarchy.....	67
6.4.2.3	Picture Paths.....	67
6.4.2.4	Picture object attributes.....	68
6.4.2.5	Named program calls.....	68
6.4.3	Predefined VS object, window and picture function methods.....	69
6.4.3.1	_FLAG_FOR_EXECUTION(name, program [, delay]).....	69
6.4.3.2	_QUEUE_FOR_EXECUTION(program [, delay]).....	69
6.4.4	Predefined VS object, window and picture function attributes.....	70
6.4.4.1	_ATTRIBUTE_NAMES.....	70
6.4.4.2	_CHILD_OBJECTS.....	70
6.4.4.3	_COMPILED.....	70
6.4.4.4	_FILE_REVISION.....	71
6.4.4.5	_OBJECT_CLASS.....	71
6.4.4.6	_OBJECT_NAME.....	71
6.4.4.7	_OBJECT_PATH.....	71
6.4.4.8	_SG_GEOMETRY.....	71
6.4.4.9	_SOURCE_FILE_NAME.....	72
6.4.4.10	_VARIABLE_NAMES.....	72
6.5	Files.....	72
6.5.1	File naming.....	72
6.5.2	Text files.....	73
6.5.3	Binary files.....	74
6.5.4	Keyed files.....	74
6.5.4.1	General.....	74
6.5.4.2	Implementation.....	75
6.5.4.3	Use.....	75
6.6	SCIL databases.....	76
6.6.1	General.....	76
6.6.2	Versions.....	76
6.6.3	Access.....	77
6.6.4	Use.....	77
<b>Section 7</b>	<b>Variables.....</b>	<b>79</b>
7.1	General.....	79
7.1.1	Variable names.....	79
7.1.2	Scope of variables.....	79
7.2	Local variables.....	79
7.3	Global variables.....	80
7.3.1	SCIL contexts.....	80
7.4	Using variables.....	81
7.4.1	Variable assignment.....	81
7.4.2	Examples.....	81

7.4.3	Using variables in expressions.....	81
7.4.4	Examples.....	82
7.4.5	Variable expansion .....	82
7.4.6	Examples.....	82
7.5	Predefined picture variables.....	82
7.5.1	Examples.....	83
<b>Section 8</b>	<b>Expressions.....</b>	<b>85</b>
8.1	General principles.....	85
8.1.1	Use.....	85
8.1.2	Composition.....	85
8.1.3	Operands.....	85
8.1.4	Operators.....	86
8.2	Arithmetical operators.....	86
8.2.1	Use.....	86
8.2.2	Operators.....	86
8.2.3	Priority order.....	86
8.2.4	Compatibility rules.....	87
8.2.5	Examples.....	89
8.3	Relational operators.....	89
8.3.1	Use.....	89
8.3.2	Operators.....	89
8.3.3	Compatibility rules.....	90
8.3.4	Examples.....	90
8.4	Logical operators.....	91
8.4.1	Use.....	91
8.4.2	Operators.....	91
8.4.3	Compatibility rules.....	91
8.4.4	Examples.....	91
8.5	SCIL Data Derivation Language (SDDL).....	91
8.5.1	General.....	91
8.5.2	Relation to SCIL language.....	92
8.5.3	Context variables.....	92
8.5.4	SDDL expression properties.....	92
8.5.5	Examples.....	92
<b>Section 9</b>	<b>SCIL statements.....</b>	<b>95</b>
9.1	General.....	95
9.1.1	Types of SCIL statements.....	95
9.1.2	Arguments.....	95
9.1.3	Overview.....	96
9.2	General SCIL statements.....	98
9.2.1	Basic SCIL statements.....	98
9.2.1.1	[@]name[component]* = value.....	98
9.2.1.2	#ARGUMENT name [,name]*.....	98
9.2.1.3	#BLOCK[statement]* #BLOCK_END.....	99

9.2.1.4	#CASE value, [ #WHEN selector statement]*, [ #OTHERWISE statement], #CASE_END.....	99
9.2.1.5	#DO program.....	100
9.2.1.6	#ERROR IGNORE, #ERROR CONTINUE, #ERROR STOP, #ERROR EVENT.....	101
9.2.1.7	#ERROR RAISE [status].....	101
9.2.1.8	#IF condition1 #THEN statement1, [ #ELSE_IF condition2 #THEN statement2]*, [ #ELSE statement3].....	102
9.2.1.9	#LOCAL name [= value] [,name [= value]]*.....	102
9.2.1.10	#LOOP [condition], [statement]*, #LOOP_END [max].....	103
9.2.1.11	#LOOP_WITH var = low .. high, [statement]*, #LOOP_END.....	105
9.2.1.12	#LOOP_EXIT.....	105
9.2.1.13	#ON event [statement].....	105
9.2.1.14	#ON ERROR [statement].....	106
9.2.1.15	#ON KEY_ERROR [statement].....	106
9.2.1.16	#PAUSE interval.....	107
9.2.1.17	#RETURN [value].....	107
9.2.1.18	#SET_TIME time.....	107
9.2.2	Application and system object commands.....	108
9.2.2.1	#CREATE object [=attributes].....	108
9.2.2.2	#DELETE object.....	108
9.2.2.3	#EXEC object [(variable_list)].....	109
9.2.2.4	#EXEC_AFTER delay object [(variable_list)].....	109
9.2.2.5	#GET object.....	110
9.2.2.6	#INIT_QUERY n [condition].....	110
9.2.2.7	#MODIFY object = attributes.....	111
9.2.2.8	#SEARCH n apl type order [start [condition]].....	112
9.2.2.9	#SET object_attribute [= value].....	113
9.2.3	Printout commands.....	114
9.2.3.1	#LIST printer object [(variable list)].....	114
9.2.3.2	#PRINT printer picture [(variable list)].....	115
9.2.4	Path commands.....	116
9.2.4.1	#PATH name [dir [, dir]*], #PATH name + dir [, dir]*, #PATH name - [dir [, dir]*].....	116
9.2.4.2	#REP_LIB library [file [, file]*], #REP_LIB library + [file [, file]*], #REP_LIB library - [file [, file]*].....	117
9.2.5	File handling commands.....	118
9.2.5.1	#CLOSE_FILE n .....	118
9.2.5.2	#CREATE_FILE n apl file keylength.....	119
9.2.5.3	#DELETE_FILE apl file.....	119
9.2.5.4	#OPEN_FILE n apl file keylength.....	119
9.2.5.5	#READ n key data1 [data2].....	120
9.2.5.6	#READ_KEYS n keys [key1 [key2]].....	120
9.2.5.7	#READ_NEXT n key data1 [data2].....	121
9.2.5.8	#READ_PREV n key data1 [data2].....	121
9.2.5.9	#REMOVE n key.....	121
9.2.5.10	#RENAME_FILE apl old new .....	122
9.2.5.11	#WRITE n data1 [data2].....	122
9.3	Visual SCIL commands.....	123

9.3.1	Loading, creating and deleting Visual SCIL objects.....	123
9.3.1.1	.CREATE object = type [(attribute = value [,attribute = value]*)].....	123
9.3.1.2	.DELETE object.....	124
9.3.1.3	.LOAD object = type(, file , name [, attribute = value]*). .....	124
9.3.2	Handling Visual SCIL attributes and methods.....	125
9.3.2.1	[object].method [(argument [,argument]*)].....	125
9.3.2.2	.MODIFY object = list.....	125
9.3.2.3	.SET [object].attribute[component]* = value.....	126
9.4	Picture handling commands.....	126
9.4.1	General picture handling commands.....	126
9.4.1.1	!CLOSE.....	126
9.4.1.2	!FAST_PIC[picture], !FAST_PIC-[picture], !FAST_PIC[-][picture], [-][picture], .....	127
9.4.1.3	!INT_PIC.....	128
9.4.1.4	!LAST_PIC.....	128
9.4.1.5	!NEW_PIC picture.....	128
9.4.1.6	!RECALL_PIC.....	129
9.4.1.7	!RESTORE.....	129
9.4.1.8	!STORE_PIC.....	129
9.4.1.9	!UPDATE interval.....	129
9.4.2	Window handling commands.....	130
9.4.2.1	!ERASE [picture path]window.....	130
9.4.2.2	!SHOW [picture path]window [expression].....	131
9.4.2.3	!SHOW_BACK [picture path]window [expression].....	131
9.4.2.4	!WIN_BG_COLOR [picture path]window color.....	132
9.4.2.5	!WIN_CREATE [picture path]window.....	132
9.4.2.6	!WIN_INPUT [picture path]window expression.....	133
9.4.2.7	!WIN_LEVEL [picture path]window level.....	133
9.4.2.8	!WIN_NAME [picture path]window.....	134
9.4.2.9	!WIN_PIC [picture path]window picture.....	134
9.4.2.10	!WIN_POS [picture path]window pos.....	135
9.4.2.11	!WIN REP [picture path]window representation.....	135
9.4.3	Input commands.....	136
9.4.3.1	!CSR_LEFT, !CSR_RIGHT, !CSR_BOL, !CSR_EOL.....	136
9.4.3.2	!ENTER.....	136
9.4.3.3	!INPUT_POS var.....	137
9.4.3.4	!INPUT_VAR [picture path]window variable max_length.....	137
9.4.3.5	!RUBOUT, !RUBOUT_CUR, !RUBOUT_BOL, !RUBOUT_EOL.....	138
9.4.4	Miscellaneous picture commands.....	138
9.4.4.1	!SEND_PIC device number.....	138
9.4.4.2	!RESET.....	139
<b>Section 10</b>	<b>Functions.....</b>	<b>141</b>
10.1	General.....	142
10.1.1	Function calls.....	142
10.1.2	Overview.....	143
10.2	Generic functions.....	156

10.2.1	CASE(selector, w <sub>1</sub> , v <sub>1</sub> [, w <sub>i</sub> , v <sub>i</sub> ]* [, "OTHERWISE", v <sub>0</sub> ]).....	156
10.2.2	CHOOSE(v), CHOOSE(v <sub>1</sub> , v <sub>2</sub> , [v <sub>i</sub> ]*). ....	156
10.2.3	DATA_TYPE(expression).....	157
10.2.4	DUMP(data [,line_length]).....	157
10.2.5	ELEMENT_LENGTH(vl).....	157
10.2.6	EQUAL(v1, v2, [,status_handling [,case_policy]]).....	158
10.2.7	EVALUATE(expression).....	158
10.2.8	GET_STATUS(data).....	159
10.2.9	IF(condition, truevalue, falsevalue).....	159
10.2.10	LENGTH (arg).....	159
10.2.11	SET_STATUS(source, status).....	160
10.2.12	TYPE_CAST(source, type).....	161
10.3	Arithmetic functions.....	161
10.3.1	ABS(arg).....	161
10.3.2	ARCCOS(arg).....	162
10.3.3	ARCSIN(arg).....	162
10.3.4	ARCTAN(arg).....	162
10.3.5	COS(arg).....	162
10.3.6	EVEN(arg).....	162
10.3.7	EXP(arg).....	162
10.3.8	HIGH_PRECISION_ADD(n1 [,n]*). ....	163
10.3.9	HIGH_PRECISION_DIV(n1, n2).....	163
10.3.10	HIGH_PRECISION_MUL(n1, n2).....	163
10.3.11	HIGH_PRECISION_SHOW(n [,decimals]).....	164
10.3.12	HIGH_PRECISION_SUB(n1, n2).....	164
10.3.13	HIGH_PRECISION_SUM(v).....	164
10.3.14	LN(arg).....	165
10.3.15	MAX(arg1 [,arg]*). ....	165
10.3.16	MIN(arg1 [,arg]*). ....	165
10.3.17	ODD(arg).....	166
10.3.18	RANDOM(n1, n2).....	166
10.3.19	ROUND(arg [,decimals]).....	167
10.3.20	SET_RANDOM_SEED(seed).....	167
10.3.21	SIN(arg).....	167
10.3.22	SQRT(arg).....	167
10.3.23	TRUNC(arg [,decimals]).....	167
10.4	Time functions.....	168
10.4.1	Qualified time.....	168
10.4.2	CLOCK.....	169
10.4.3	DATE[(time [, "FULL"])].....	170
10.4.4	DAY[(time)].....	170
10.4.5	DOW[(time)].....	170
10.4.6	DOY[(time)].....	170
10.4.7	HOD[(time)].....	170
10.4.8	HOUR[(time)].....	171
10.4.9	HOY[(time)].....	171

10.4.10	HR_CLOCK.....	171
10.4.11	LOCAL_TIME.....	171
10.4.12	LOCAL_TIME_ADD(time, s [,ms]).....	171
10.4.13	LOCAL_TIME_INFORMATION[(time)].....	172
10.4.14	LOCAL_TIME_INTERVAL(from, to).....	172
10.4.15	LOCAL_TO_SYS_TIME(time).....	173
10.4.16	LOCAL_TO_UTC_TIME(time).....	173
10.4.17	MINUTE[(time)].....	173
10.4.18	MONTH[(time)].....	173
10.4.19	PACK_TIME(year, month, day, hour, minute, second).....	173
10.4.20	SECOND[(time)].....	174
10.4.21	SET_CLOCK(time).....	174
10.4.22	SET_LOCAL_TIME(time).....	174
10.4.23	SET_SYS_TIME(time).....	174
10.4.24	SET_UTC_TIME(time).....	175
10.4.25	SYS_TIME.....	175
10.4.26	SYS_TIME_ADD(time, s [,ms]).....	175
10.4.27	SYS_TIME_INTERVAL(from, to).....	176
10.4.28	SYS_TO_LOCAL_TIME(time).....	176
10.4.29	SYS_TO_UTC_TIME(time).....	176
10.4.30	TIME[(time [, "FULL"])].....	177
10.4.31	TIME_SCAN(string [,resolution [,option1 [,option2]]]).....	177
10.4.32	TIME_ZONE_RULES[(rule)].....	178
10.4.33	TIMEMS[(time [,msecs] [, "FULL"])].....	180
10.4.34	TIMES[(time [, "FULL"])].....	180
10.4.35	TOD[(time)].....	181
10.4.36	TODMS[(time [,msecs])].....	181
10.4.37	TODS[(time)].....	181
10.4.38	UTC_TIME.....	181
10.4.39	UTC_TIME_ADD(time, s [,ms]).....	181
10.4.40	UTC_TIME_INTERVAL(from, to).....	182
10.4.41	UTC_TO_LOCAL_TIME(time).....	182
10.4.42	UTC_TO_SYS_TIME(time).....	182
10.4.43	WEEK(time)].....	182
10.4.44	YEAR(time)].....	183
10.5	String functions.....	183
10.5.1	ASCII(n).....	183
10.5.2	ASCII_CODE(c).....	184
10.5.3	BCD_TO_INTEGER(bcd).....	184
10.5.4	BIN(b).....	185
10.5.5	BIN_SCAN(string).....	185
10.5.6	BIT_SCAN(string).....	185
10.5.7	CAPITALIZE(text).....	185
10.5.8	COLLECT(v, delimiter).....	186
10.5.9	DEC(value [,length [,decimals]]).....	186
10.5.10	DEC_SCAN(string).....	186

10.5.11	EDIT(text, key).....	187
10.5.12	HEX(n).....	187
10.5.13	HEX_SCAN(string).....	187
10.5.14	INTEGER_TO_BCD(int [,digits]).....	188
10.5.15	JOIN(delimiter [, a]*).....	189
10.5.16	LOCATE(string1, string2 [, "ALL"]).....	189
10.5.17	LOWER_CASE(text).....	190
10.5.18	OCT(n).....	190
10.5.19	OCT_SCAN(string).....	190
10.5.20	PACK_STR(source, type [,length [,byte_order]]).....	190
10.5.21	PAD([string, ]filler, length).....	191
10.5.22	REPLACE(text, string, new_string).....	192
10.5.23	SEPARATE(text, delimiter).....	192
10.5.24	SUBSTR(string, start [,length]).....	192
10.5.25	UNPACK_STR(source [,length [,byte_order]]).....	193
10.5.26	UPPER_CASE(text).....	193
10.6	Bit functions.....	194
10.6.1	BIT(a, b).....	194
10.6.2	BIT_AND(a1, a2).....	194
10.6.3	BIT_CLEAR(a [,b]*).....	194
10.6.4	BIT_COMPL(a).....	195
10.6.5	BIT_MASK([b1 [,b]]*).....	195
10.6.6	BIT_OR(a1, a2).....	195
10.6.7	BIT_SET(a [,b]*).....	196
10.6.8	BIT_STRING(length [,b]*).....	196
10.6.9	BIT_XOR(a1, a2).....	196
10.7	Vector handling functions.....	197
10.7.1	APPEND(v, data).....	197
10.7.2	BINARY_SEARCH(v, value).....	197
10.7.3	CLASSIFY(v, n, low, high).....	198
10.7.4	CUMULATE(v).....	198
10.7.5	DELETE_ELEMENT(v, index [,index2]).....	198
10.7.6	FIND_ELEMENT(v, value [,start_index [,case_policy]]).....	199
10.7.7	HIGH(v), LOW(v).....	199
10.7.8	HIGH_INDEX(v), LOW_INDEX(v).....	200
10.7.9	INSERT_ELEMENT(v, pos, contents).....	200
10.7.10	INTERP(v, x).....	201
10.7.11	INVERSE(v, n, low, high).....	201
10.7.12	MEAN(v).....	202
10.7.13	PICK(v, indices).....	202
10.7.14	REMOVE_DUPLICATES(v [,status_handling [,case_policy]]).....	203
10.7.15	REVERSE(v).....	203
10.7.16	SELECT(source, condition [,wildcards]).....	203
10.7.17	SHUFFLE(n).....	205
10.7.18	SORT(v, [start, [length]]).....	205
10.7.19	SPREAD(v, indices, new_value).....	205

10.7.20	SUM(v), SUM_POS(v), SUM_NEG(v).....	206
10.7.21	TREND(v, n).....	207
10.7.22	VECTOR {[element1 [,element]*]}.....	207
10.8	List handling functions.....	207
10.8.1	ATTRIBUTE_EXISTS(list, attribute).....	207
10.8.2	DELETE_ATTRIBUTE(list, attribute).....	207
10.8.3	LIST([attribute = expression, [attribute = expression]*]).....	208
10.8.4	LIST_ATTR(list).....	208
10.8.5	MERGE_ATTRIBUTES(left, right).....	208
10.9	Functions related to program execution.....	209
10.9.1	ARGUMENT(n).....	209
10.9.2	ARGUMENT_COUNT.....	209
10.9.3	ARGUMENTS.....	209
10.9.4	DO(program [,a]*).....	209
10.9.5	ERROR_STATE.....	210
10.9.6	MEMORY_USAGE(keyword, arg).....	210
10.9.7	OPS_CALL(command [,nowait]), OPS_CALL(command [,option1 [,option2]]).....	211
10.9.8	OPS_PROCESS(command [,directory [,option1 [,option2]]]).....	211
10.9.9	REVISION_COMPATIBILITY(issue [,enable]).....	212
10.9.10	STATUS.....	213
10.9.11	VARIABLE_NAMES.....	213
10.10	Functions related to the run-time environment.....	213
10.10.1	AEP_PROGRAMS(apl).....	213
10.10.2	CONSOLE_OUTPUT(text [,severity [,category]]).....	213
10.10.3	ENVIRONMENT(variable).....	214
10.10.4	IP_PROGRAMS.....	214
10.10.5	MEMORY_POOL_USAGE(pool).....	214
10.10.6	OPS_NAME{[major [,minor]]}.....	215
10.10.7	REGISTRY(function, key, value_name).....	216
10.10.8	SCIL_HOST.....	216
10.11	Functions related to the programming environment.....	218
10.11.1	COMPILE(source).....	218
10.11.2	MAX_APPLICATION_NUMBER.....	218
10.11.3	MAX_BIT_STRING_LENGTH.....	218
10.11.4	MAX_BYTE_STRING_LENGTH.....	218
10.11.5	MAX_INTEGER.....	218
10.11.6	MAX_LINK_NUMBER.....	218
10.11.7	MAX_LIST_ATTRIBUTE_COUNT.....	219
10.11.8	MAX_MONITOR_NUMBER.....	219
10.11.9	MAX_NODE_NUMBER.....	219
10.11.10	MAX_OBJECT_NAME_LENGTH.....	219
10.11.11	MAX_PICTURE_NAME_LENGTH.....	219
10.11.12	MAX_PRINTER_NUMBER.....	219
10.11.13	MAX_PROCESS_OBJECT_INDEX.....	219
10.11.14	MAX_REPRESENTATION_NAME_LENGTH.....	219
10.11.15	MAX_STATION_NUMBER.....	220

10.11.16	MAX_STATION_TYPE_NUMBER.....	220
10.11.17	MAX_TEXT_LENGTH.....	220
10.11.18	MAX_VECTOR_LENGTH.....	220
10.11.19	MAX_WINDOW_NAME_LENGTH.....	220
10.11.20	MIN_INTEGER.....	220
10.11.21	OBJECT_ATTRIBUTE_INFO(apl, type [,subtype [,selection]]).....	220
10.11.22	STATUS_CODE(mnemonic).....	222
10.11.23	STATUS_CODE_NAME(code).....	223
10.11.24	VALIDATE(as, string).....	223
10.11.25	VALIDATE_OBJECT_ADDRESS(apl, pt, un, oa [,subaddress] [, self]).....	223
10.12	Language functions.....	224
10.12.1	Language identifiers.....	225
10.12.2	The language of the SCIL context.....	227
10.12.3	Text databases.....	227
10.12.4	SET_LANGUAGE(language).....	228
10.12.5	TRANSLATE(text [,language]).....	228
10.12.6	TRANSLATION(id [,language]).....	229
10.13	Error tracing functions.....	229
10.13.1	SCIL_LINE_NUMBER.....	229
10.13.2	TRACE_BEGIN(filename [,append] [,time_tags] [,no_cyclics]).....	230
10.13.3	TRACE_END.....	230
10.13.4	TRACE_PAUSE.....	230
10.13.5	TRACE_RESUME.....	230
10.14	Database functions.....	230
10.14.1	General object listing functions.....	230
10.14.2	APPLICATION_OBJECT_ATTRIBUTES(apl, type, objects, attributes).....	230
10.14.3	APPLICATION_OBJECT_COUNT(apl, type [,order [,direction [, start [,condition]]]])	231
10.14.4	APPLICATION_OBJECT_EXISTS(apl, type, name [,condition [,verbosity]]).....	232
10.14.5	APPLICATION_OBJECT_LIST(apl, type [,order [,direction [,start [,condition [,attributes [,max]]]]]]).....	232
10.14.6	APPLICATION_OBJECT_SELECT(apl, type, names, condition [,verbosity]).....	235
10.14.7	BASE_SYSTEM_OBJECT_LIST(type [,condition [,attributes [,apl]]]).....	236
10.14.8	Object maintenance functions.....	237
10.14.9	FETCH(apl, type, name [,index]).....	237
10.14.10	NEXT(n), PREV(n).....	238
10.14.11	PHYS_FETCH(apl, unit, address [,bit_address]).....	238
10.14.12	Alarm list functions.....	238
10.14.13	APPLICATION_ALARM_COUNT(apl [, filter]).....	238
10.14.14	APPLICATION_ALARM_LIST(apl, lists [,attributes [, order [, filter [, max_count]]]])	239
10.14.15	Data object functions.....	241
10.14.16	DATA_FETCH(apl, name, index1 [,step [,count]]), DATA_FETCH(apl, name, time1, time2 [,step [,shift]]), DATA_FETCH(apl, name, time1 [,step [,count [,shift]]]), DATA_FETCH(apl, name, indices).....	241
10.14.17	DATA_STORE(apl, name, data, index1 [,step]), DATA_STORE(apl, name, data, indices).....	242
10.14.18	Process object query functions.....	243
10.14.19	END_QUERY.....	243

10.14.20	PROD_QUERY(n).....	244
10.14.21	History database functions.....	245
10.14.22	HISTORY_DATABASE_MANAGER("OPEN" [,apl]).....	245
10.14.23	HISTORY_DATABASE_MANAGER("CLOSE", session).....	245
10.14.24	HISTORY_DATABASE_MANAGER("SET_PERIOD", session, begin [,end]).....	245
10.14.25	HISTORY_DATABASE_MANAGER("SET_DIRECTORY", session, directory).....	246
10.14.26	HISTORY_DATABASE_MANAGER("SET_WINDOW", session, begin, end).....	246
10.14.27	HISTORY_DATABASE_MANAGER("SET_ORDER", session, order).....	247
10.14.28	HISTORY_DATABASE_MANAGER("SET_DIRECTION", session, direction).....	247
10.14.29	HISTORY_DATABASE_MANAGER("SET_TIMEOUT", session, timeout).....	248
10.14.30	HISTORY_DATABASE_MANAGER("SET_CONDITION", session, condition).....	248
10.14.31	HISTORY_DATABASE_MANAGER("SET_ATTRIBUTES", session, attributes).....	248
10.14.32	HISTORY_DATABASE_MANAGER("GET_PARAMETERS", session).....	249
10.14.33	HISTORY_DATABASE_MANAGER("QUERY", session, count [,start]).....	249
10.14.34	HISTORY_DATABASE_MANAGER("READ", session, event).....	250
10.14.35	HISTORY_DATABASE_MANAGER("SET_COMMENT", session, event, comment).....	250
10.14.36	HISTORY_DATABASE_MANAGER("WRITE", session, data).....	251
10.14.37	Name hierarchy function.....	251
10.14.38	NAME_HIERARCHY(names, order, syntax [, arg4 [, arg5]]).....	251
10.14.39	Mapping functions.....	255
10.14.40	LOGICAL_MAPPING(otype, number [, apl]).....	255
10.14.41	PHYSICAL_MAPPING(otype, number [, apl]).....	256
10.15	Network Topology Functions.....	256
10.15.1	NETWORK_TOPOLOGY_MANAGER(subfunction [, arg]* [, apl]).....	256
10.15.2	NETWORK_TOPOLOGY_MANAGER("SCHEMAS").....	257
10.15.3	NETWORK_TOPOLOGY_MANAGER("SCHEMA", schema).....	257
10.15.4	NETWORK_TOPOLOGY_MANAGER("LEVELS", schema).....	257
10.15.5	NETWORK_TOPOLOGY_MANAGER("SET_LEVELS", schema, levels).....	258
10.15.6	NETWORK_TOPOLOGY_MANAGER("MODELS").....	258
10.15.7	NETWORK_TOPOLOGY_MANAGER("MODEL", model).....	258
10.15.8	NETWORK_TOPOLOGY_MANAGER("START", model), NETWORK_TOPOLOGY_MANAGER("STOP", model).....	258
10.15.9	NETWORK_TOPOLOGY_MANAGER("VALIDATE", modeldata).....	259
10.15.10	NETWORK_TOPOLOGY_MANAGER("IMPORT", modeldata).....	259
10.15.11	NETWORK_TOPOLOGY_MANAGER("EXPORT", model).....	259
10.15.12	NETWORK_TOPOLOGY_MANAGER("DELETE", model).....	259
10.16	File handling functions.....	260
10.16.1	DATA_MANAGER(function [argument]*).....	260
10.16.2	DATA_MANAGER("CREATE", file).....	260
10.16.3	DATA_MANAGER("OPEN", file).....	260
10.16.4	DATA_MANAGER("COPY", handle, new_file [, version]).....	260
10.16.5	DATA_MANAGER("CLOSE", handle).....	261
10.16.6	DATA_MANAGER("LIST_SECTIONS", handle).....	261
10.16.7	DATA_MANAGER("CREATE_SECTION", handle, section).....	261
10.16.8	DATA_MANAGER("DELETE_SECTION", handle, section).....	261
10.16.9	DATA_MANAGER("GET", handle, section [,component]*).....	262

10.16.10	DATA_MANAGER("PUT", handle, section, data [,component]*)	262
10.16.11	DATA_MANAGER("DELETE", handle, section, [,component]*)	263
10.16.12	DELETE_PARAMETER(file, section [,key])	263
10.16.13	FILE_LOCK_MANAGER(function, file)	263
10.16.14	KEYED_FILE_MANAGER(function, file [,output_file [,key_size] [,version]])	264
10.16.15	PARSE_FILE_NAME(name [,file])	265
10.16.16	PATH(name)	266
10.16.17	PATHS(level)	266
10.16.18	READ_BYTES(file [,start [,length]])	266
10.16.19	READ_COLUMNS(file, pos, width [,start [,count]])	267
10.16.20	READ_PARAMETER(file, section, key [,default])	267
10.16.21	READ_TEXT(file [,start [,number]])	268
10.16.22	REP_LIB(name)	268
10.16.23	REP_LIBS(level)	269
10.16.24	SHADOW_FILE(file_name [, "DELETED"])	269
10.16.25	TEXT_READ(file [,start [,number]])	269
10.16.26	WRITE_BYTES(file, data [,append])	270
10.16.27	WRITE_COLUMNS(file, pos, width, data [,append] [,encoding])	270
10.16.28	WRITE_PARAMETER(file, section, key, value [, encoding])	271
10.16.29	WRITE_TEXT(file, text [,append] [,encoding])	271
10.17	File management functions	272
10.17.1	Calling syntax	273
10.17.2	Compatibility	273
10.17.3	DRIVE_MANAGER	273
10.17.4	DRIVE_MANAGER("LIST")	274
10.17.5	DRIVE_MANAGER("EXISTS", drive)	274
10.17.6	DRIVE_MANAGER("GET_DEFAULT")	274
10.17.7	DRIVE_MANAGER("GET_ATTRIBUTES", tag)	274
10.17.8	DIRECTORY_MANAGER	274
10.17.9	DIRECTORY_MANAGER("LIST", directory [,filter [,recursion] [,hidden]])	275
10.17.10	DIRECTORY_MANAGER("CREATE", directory [,recursion])	275
10.17.11	DIRECTORY_MANAGER("DELETE", directory)	275
10.17.12	DIRECTORY_MANAGER("DELETE_CONTENTS", directory [,filter [,subdirectories]])	275
10.17.13	DIRECTORY_MANAGER("EXISTS", directory)	276
10.17.14	DIRECTORY_MANAGER("COPY", source, target)	276
10.17.15	DIRECTORY_MANAGER("COPY_CONTENTS", source, target [,filter [,subdirectories [,overwrite]]])	276
10.17.16	DIRECTORY_MANAGER("MOVE", directory, target)	277
10.17.17	DIRECTORY_MANAGER("RENAME", directory, name)	277
10.17.18	DIRECTORY_MANAGER("GET_ATTRIBUTES", directory)	277
10.17.19	FILE_MANAGER	278
10.17.20	FILE_MANAGER("LIST", directory [,filter [,recursion] [,hidden]])	278
10.17.21	FILE_MANAGER("DELETE", file)	278
10.17.22	FILE_MANAGER("EXISTS", file)	278
10.17.23	FILE_MANAGER("COPY", source, target [,overwrite])	278

10.17.24	FILE_MANAGER("MOVE", file, target).....	279
10.17.25	FILE_MANAGER("RENAME", file, name).....	279
10.17.26	FILE_MANAGER("GET_ATTRIBUTES", file).....	279
10.17.27	Auxiliary functions.....	280
10.17.28	FM_APPLICATION_DIRECTORY[(path)].....	280
10.17.29	FM_APPLICATION_FILE(path).....	280
10.17.30	FM_COMBINE(tag1 [,tagi]*, tagn).....	280
10.17.31	FM_COMBINE_NAME(name, extension).....	280
10.17.32	FM_DIRECTORY(path [,check]).....	281
10.17.33	FM_DRIVE(name [,check]).....	281
10.17.34	FM_EXTRACT(tag, component).....	281
10.17.35	FM_FILE(path [,check]).....	282
10.17.36	FM_REPRESENT(tag [,option]*).....	282
10.17.37	FM_SCIL_DIRECTORY(name [,check]).....	282
10.17.38	FM_SCIL_FILE(name [,option] [,option]).....	283
10.17.39	FM_SCIL_REPRESENT(tag [,case]).....	283
10.17.40	FM_SPLIT_NAME(file).....	283
10.18	Communication functions.....	283
10.18.1	SPACOM(message).....	283
10.18.2	TIMEOUT(milliseconds).....	284
10.19	CSV (Comma Separated Value) functions.....	284
10.19.1	CSV_TO_SCIL(csv, start, field_info [,option]).....	285
10.19.2	SCIL_TO_CSV(data [,option]*).....	286
10.20	DDE client functions.....	286
10.20.1	DDE_CONNECT(service, topic).....	287
10.20.2	DDE_DISCONNECT(connection_id).....	288
10.20.3	DDE_REQUEST(connection_id, item [,timeout]).....	289
10.20.4	DDE_POKE(connection_id, item, value [,timeout]).....	289
10.20.5	DDE_EXECUTE(connection_id, statement [,timeout]).....	290
10.21	DDE server functions.....	290
10.21.1	DDE_VECTOR(vector, decimal_separator, list_separator).....	291
10.21.2	DDE_REAL(real, separator).....	291
10.22	ODBC functions.....	292
10.22.1	SQL_CONNECT(source, user, password).....	292
10.22.2	SQL_DISCONNECT(connection_id).....	293
10.22.3	SQL_EXECUTE(connection_id, SQLstring [,timeout]).....	293
10.22.4	SQL_FETCH(statement_id).....	294
10.22.5	SQL_FREE_STATEMENT(statement_id).....	295
10.22.6	SQL_BEGIN_TRANSACTION(connection_id).....	295
10.22.7	SQL_COMMIT(connection_id).....	296
10.22.8	SQL_ROLLBACK(connection_id).....	296
10.23	OPC Name Database functions.....	297
10.23.1	OPC_NAME_MANAGER(function, apl [,argument]*).....	298
10.23.2	OPC_NAME_MANAGER("LIST", apl).....	298
10.23.3	OPC_NAME_MANAGER("PUT", apl, name, definition).....	298
10.23.4	OPC_NAME_MANAGER("GET", apl, name).....	298

10.23.5	OPC_NAME_MANAGER("DELETE", apl, name).....	299
10.24	OPC functions.....	299
10.24.1	OPC_AE_ACKNOWLEDGE(apl, ln, ix, ack_id [, comment [, cookie, active_time]]).....	299
10.24.2	OPC_AE_NAMESPACE(nodenr [, clsid1 [, root]]) OPC_AE_NAMESPACE(nodename, clsid2 [, root [, user, password]]).....	300
10.24.3	OPC_AE_REFRESH(apl, unit).....	301
10.24.4	OPC_AE_SERVERS(nodenr), OPC_AE_SERVERS(nodename [, user, password]).....	301
10.24.5	OPC_AE_VALIDATE(apl, unit).....	302
10.24.6	OPC_DA_NAMESPACE(nodenr [, clsid1 [, root]]) OPC_DA_NAMESPACE(nodename, clsid2 [, root [, user, password]]).....	303
10.24.7	OPC_DA_REFRESH(apl, unit, group [, wait]).....	305
10.24.8	OPC_DA_SERVERS(nodenr), OPC_DA_SERVERS(nodename [, user, password]).....	305
10.25	RTU functions.....	306
10.25.1	RTU_ADDR(key).....	306
10.25.2	RTU_AINT(i).....	306
10.25.3	RTU_AREAL(r).....	307
10.25.4	RTU_ATIME [(t [,msec])].....	307
10.25.5	RTU_BIN(h).....	307
10.25.6	RTU_HEXASC(b).....	307
10.25.7	RTU_INT(a).....	308
10.25.8	RTU_KEY(oa).....	308
10.25.9	RTU_MSEC(atime).....	308
10.25.10	RTU_OA(type, ba).....	308
10.25.11	RTU_REAL(a).....	308
10.25.12	RTU_TIME(a).....	309
10.26	Printout functions.....	309
10.26.1	PRINT_TRANSPARENT(data [,log]).....	309
10.26.2	PRINTER_SET.....	312
10.27	User session functions.....	312
10.27.1	SET_EVENT_LIST_USER_NAME(name).....	312
10.27.2	USM_ADDRESS.....	313
10.27.3	USM_AOR_DATA.....	313
10.27.4	USM_AUTHORIZATION_LEVEL(group).....	313
10.27.5	USM_AUTHORIZATION_LEVEL_FOR_OBJECT(group, object_name, object_index).....	313
10.27.6	USM_AUTHORIZATIONS.....	314
10.27.7	USM_CHANGE_PASSWORD(old_password, new_password).....	314
10.27.8	USM_IS_NEW_APPLICATION.....	314
10.27.9	USM_LOGIN(name, password).....	315
10.27.10	USM_LOGOUT.....	315
10.27.11	USM_PASSWORD_CHANGE.....	315
10.27.12	USM_PASSWORD_POLICY.....	316
10.27.13	USM_SELECT_ROLE(role).....	316
10.27.14	USM_SESSION_ATTRIBUTES.....	317
10.27.15	USM_SESSION_ID.....	317
10.27.16	USM_SESSIONS.....	317
10.27.17	USM_USER_LANGUAGE.....	318

10.27.18	USM_USER_NAME.....	318
10.27.19	USM_USER_ROLE.....	318
10.27.20	USM_USER_ROLES.....	318
10.27.21	USM_USER_SESSION_DATA.....	318
10.28	Miscellaneous functions.....	318
10.28.1	ADD_INTERLOCKED(object, index, amount).....	318
10.28.2	AUDIO_ALARM(alarm_class, on_or_off).....	319
10.28.3	SCALE(v, scale_object [,direction]).....	319
10.28.4	UNLOCK_PICTURE(picture).....	320

<b>Section 11</b>	<b>Graphics primitives.....</b>	<b>321</b>
11.1	Introduction.....	321
11.1.1	Graphics contexts.....	321
11.1.2	Graphics canvas.....	321
11.1.3	Miscellaneous.....	322
11.2	Full graphics SCIL commands.....	322
11.2.1	Drawing graphical elements.....	322
11.2.1.1	.ARC [[scope,]n :] x,y, r, a1,a2 [,FILL].....	322
11.2.1.2	.BOX [[scope,]n :] x,y, width, height [,FILL].....	323
11.2.1.3	.CIRCLE [[scope,]n :] x,y, r [,FILL].....	323
11.2.1.4	.ELLIPSE [[scope,]n :] x,y, a,b [,FILL].....	323
11.2.1.5	.IMAGE x, y, w, h, filename, tag_1[, tag_2[, tag_3[, tag_4]]].....	324
11.2.1.6	.LINE [[scope,]n :] x1,y1, x2,y2.....	324
11.2.1.7	.POINT [[scope,]n :] x,y, [RELATIVE].....	325
11.2.1.8	.POLYLINE [[scope,]n :] x,y [,RELATIVE] [,FILL].....	325
11.2.1.9	.TEXT [[scope,]n :] x,y, text [,FILL] [,align].....	326
11.3	Graphics contexts.....	326
11.3.1	General.....	326
11.3.1.1	Scope of graphics contexts.....	327
11.3.1.2	Default settings.....	327
11.3.2	Defining graphics contexts.....	328
11.3.2.1	.GC [[scope,]n [=scope,]m]:[[component = value]...[,component = value]].....	328
11.3.3	Components of graphics contexts.....	328
11.3.3.1	AM      ARC_MODE.....	329
11.3.3.2	BG      BACKGROUND.....	329
11.3.3.3	CS      CAP_STYLE.....	329
11.3.3.4	DL      DASH_LIST.....	329
11.3.3.5	DO      DASH_OFFSET.....	330
11.3.3.6	FT      FONT.....	331
11.3.3.7	FG      FOREGROUND.....	331
11.3.3.8	FU      FUNCTION.....	331
11.3.3.9	JS      JOIN_STYLE.....	332
11.3.3.10	LS      LINE_STYLE.....	332
11.3.3.11	LW      LINE_WIDTH.....	332
11.3.3.12	NA      NAME.....	333
11.3.4	Colors and fonts.....	333

11.3.4.1	Colors.....	333
11.3.4.2	.COLOR [scope,] number : color [,SHARED] .....	333
11.3.4.3	COLOR([scope,]number).....	333
11.3.4.4	Fonts.....	334
11.3.4.5	.FONT [scope,] number : font.....	334
11.3.4.6	FONT([scope,] number).....	334
11.3.5	Reading graphics contexts.....	335
11.3.5.1	GC([scope,]n).....	335
11.3.5.2	COLOR_IN([scope,] n).....	336
11.3.5.3	FONT_IN([scope,] n).....	336
11.4	Graphics canvas.....	336
11.4.1	General description.....	336
11.4.2	Selecting canvas.....	336
11.4.3	.CANVAS object.....	336
11.4.4	.CANVAS ROOT.....	337
11.4.5	.CANVAS PARENT.....	337
11.4.6	.CANVAS CURRENT.....	337
11.4.7	.COORDINATE_SYSTEM coordinate_system.....	337
11.4.8	The SCIL coordinate system.....	337
11.4.9	The Visual SCIL coordinate system.....	338
11.4.10	Changing Scaling Factor.....	338
11.4.11	.SCALING [s].....	338
11.4.12	Mouse input.....	339
11.4.13	.MOUSE x, y [, button [, buttons [, RELATIVE] ] ].....	339
11.4.14	.MOUSE ON [,MOTION], .MOUSE OFF.....	339
11.4.15	.MOUSE DISCARD.....	340
11.5	Miscellaneous graphical commands.....	341
11.5.1	Storing and restoring selections.....	341
11.5.1.1	.PUSH, .POP.....	341
11.5.2	Display handling commands.....	341
11.5.2.1	.FLUSH.....	341
11.5.2.2	.PEND ON, .PEND OFF.....	341
<b>Section 12</b>	<b>SCIL programming guide.....</b>	<b>343</b>
12.1	Picture handling.....	343
12.2	Visual SCIL object handling.....	344
12.3	Program execution.....	345
12.3.1	Error Handling in Pictures.....	347
<b>Section 13</b>	<b>SCIL editor.....</b>	<b>349</b>
13.1	General.....	349
13.2	Menus.....	349
13.3	Toolbar.....	352
13.4	Opening and closing the SCIL editor.....	353
13.4.1	Opening the SCIL editor.....	353
13.4.2	Opening files.....	353
13.4.3	Creating files.....	353

13.4.4	Saving files.....	354
13.4.5	Undo operation.....	354
13.4.6	Closing the SCIL editor.....	354
13.5	Typing and editing programs and texts.....	354
13.5.1	Typing.....	354
13.5.2	Scroll feature.....	354
13.5.3	Selecting text for editing.....	355
13.5.4	Copying.....	355
13.5.5	Moving text.....	355
13.5.6	Deleting.....	355
13.5.7	Commenting.....	355
13.5.8	Indenting.....	356
13.5.9	Finding text.....	356
13.5.10	Replacing text.....	357
13.5.11	Finding blocks.....	358
13.5.12	Finding a line .....	358
13.5.13	Importing and exporting text.....	359
13.5.14	Undoing and redoing operations .....	359
13.5.15	Insert SCIL commands, functions and objects.....	359
13.5.16	Syntax checking of a SCIL program.....	360
<b>Section 14</b>	<b>SCIL compiler.....</b>	<b>363</b>
14.1	General.....	363
14.2	Performance improvement.....	363
14.3	Impact on SCIL programs.....	363
14.3.1	Programs that do not compile.....	364
14.3.2	Programs that generate run-time error.....	364
14.3.3	Programs that produce wrong results.....	365
14.3.4	Recommendations.....	365
<b>Appendix A</b>	<b>ODBC ERROR CODES.....</b>	<b>367</b>
1.1	About this appendix.....	367
1.2	General.....	367
1.3	List of codes.....	367
<b>Appendix B</b>	<b>PARAMETER FILES.....</b>	<b>369</b>
1.1	About this appendix.....	369
1.2	General.....	369
1.3	BNF description.....	369
<b>Index.....</b>		<b>371</b>



# Section 1      Copyrights

The information in this document is subject to change without notice and should not be construed as a commitment by Hitachi Power Grids. Hitachi Power Grids assumes no responsibility for any errors that may appear in this document.

In no event shall Hitachi Power Grids be liable for direct, indirect, special, incidental or consequential damages of any nature or kind arising from the use of this document, nor shall Hitachi Power Grids be liable for incidental or consequential damages arising from the use of any software or hardware described in this document.

This document and parts thereof must not be reproduced or copied without written permission from Hitachi Power Grids, and the contents thereof must not be imparted to a third party nor used for any unauthorized purpose.

The software or hardware described in this document is furnished under a license and may be used, copied, or disclosed only in accordance with the terms of such license.

© 2021 Hitachi Power Grids. All rights reserved.

## Trademarks

ABB is a registered trademark of ABB Asea Brown Boveri Ltd. Manufactured by/for a Hitachi Power Grids company. All other brand or product names mentioned in this document may be trademarks or registered trademarks of their respective holders.

## Guarantee

Please inquire about the terms of guarantee from your nearest Hitachi Power Grids representative.

## Third Party Copyright Notices

List of Third Party Copyright notices are documented in "3rd party licenses.txt" and other locations mentioned in the file in SYS600 and DMS600 installation packages.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (<https://www.openssl.org/>). This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).



## Section 2      About this manual

### 2.1    Use of symbols

This publication includes warning, caution and information symbols where appropriate to point out safety-related or other important information. It also includes tips to point out useful hints to the reader. The corresponding symbols should be interpreted as follows:



Warning icon indicates the presence of a hazard which could result in personal injury.



Caution icon indicates important information or a warning related to the concept discussed in the text. It might indicate the presence of a hazard, which could result in corruption of software or damage to equipment/property.



Information icon alerts the reader to relevant factors and conditions.



Tip icon indicates advice on, for example, how to design a project or how to use a certain function.

Although warning hazards are related to personal injury, and caution hazards are associated with equipment or property damage, it should be understood that operation of damaged equipment could, under certain operational conditions, result in degraded process performance leading to personal injury or death. Therefore, comply fully with all warnings and caution notices.

### 2.2    Intended audience

This manual is intended for installation personnel, administrators and skilled operators to support installation of the software.

### 2.3    Related documents

Name of the manual	Document ID
SYS600 10.2 Application Objects	1MRK 511 467-UEN
SYS600 10.2 System Configuration	1MRK 511 481-UEN
SYS600 10.2 Status Codes	1MRK 511 480-UEN
SYS600 10.2 System Objects	1MRK 511 482-UEN
SYS600 10.2 Visual SCIL Objects	1MRK 511 484-UEN

### 2.4    Document conventions

The following conventions are used for the presentation of material:

- The words in names of screen elements (for example, the title in the title bar of a dialog, the label for a field of a dialog box) are initially capitalized.
- Capital letters are used for file names.
- Capital letters are used for the name of a keyboard key if it is labeled on the keyboard. For example, press the CTRL key. Although the Enter and Shift keys are not labeled they are written in capital letters, for example, press ENTER.
- Lowercase letters are used for the name of a keyboard key that is not labeled on the keyboard. For example, the space bar, comma key and so on.
- Press CTRL+C indicates that the user must hold down the CTRL key while pressing the C key (in this case, to copy a selected object).
- Press ALT E C indicates that the user presses and releases each key in sequence (in this case, to copy a selected object).
- The names of push and toggle buttons are boldfaced. For example, click **OK**.
- The names of menus and menu items are boldfaced. For example, the **File** menu.
  - The following convention is used for menu operations: **Menu Name/Menu Item/Cascaded Menu Item**. For example: select **File/Open/New Project**.
  - The **Start** menu name always refers to the **Start** menu on the Windows Task Bar.
- System prompts/messages and user responses/input are shown in the Courier font. For example, if the user enters a value that is out of range, the following message is displayed: Entered value is not valid.
- The user may be told to enter the string MIF349 in a field. The string is shown as follows in the procedure: MIF349
- Variables are shown using lowercase letters: sequence name

## 2.5 Document revisions

Revision	Version number	Date	History
A	10.2	31.03.2021	New document for SYS600 10.2

# Section 3      Introduction

This section introduces the SCIL programming language, the SYS600 application engineering, and the SCIL program structure.

## 3.1      Presentation of SCIL

### 3.1.1    What is SCIL

SCIL, Supervisory Control Implementation Language, is a high level programming language especially designed for the application engineering of the supervisory control system SYS600. All SYS600 application programs, as well as most of the system configuration programs are built in SCIL.

### 3.1.2    Application engineering

In SYS600, application engineering means the composition of customized, process specific supervisory control software. The result is an application software package regarding control functions, communicating process devices, user interface, level of information, etc., that is adapted for the user's needs. A base system contains one or more application software packages named applications. Application engineering comprises of the following:

- Functional design, that is, the programming and definition of the SCADA functions, as well as other supervisory control and calculation functions.
- User interface design.

Functional design involves the definition of databases (a database = a set of connected data stored in a structured form) and the creation of SCIL programs. Each application has a process database for handling process supervision and a report database for data storage, calculations, automatic activation, etc. The databases are composed of objects named application objects.

The user interface design involves the composition of pictures and dialogs, see [Figure 1](#). Pictures are dynamic illustrations containing a static background, dynamic windows and user activated function keys. Dialogs are independent windows that may contain a wide range of user interface items, such as menus, buttons, images, notebooks and pictures. Pictures and dialogs represent two different user interface design methods. The design and programming of dialogs and dialog systems is named Visual SCIL.

These two portions of an application are interwoven with each others, and the user interface design and the functional design generally occur in parallel.

Application engineering is simplified by using the standard application software library, LIB 500, which requires a minimum of object definitions and SCIL programming. However, SCIL is found in all SYS600 applications, even in those that are built with LIB 500, because the LIB 500 standard application software is built with SCIL.

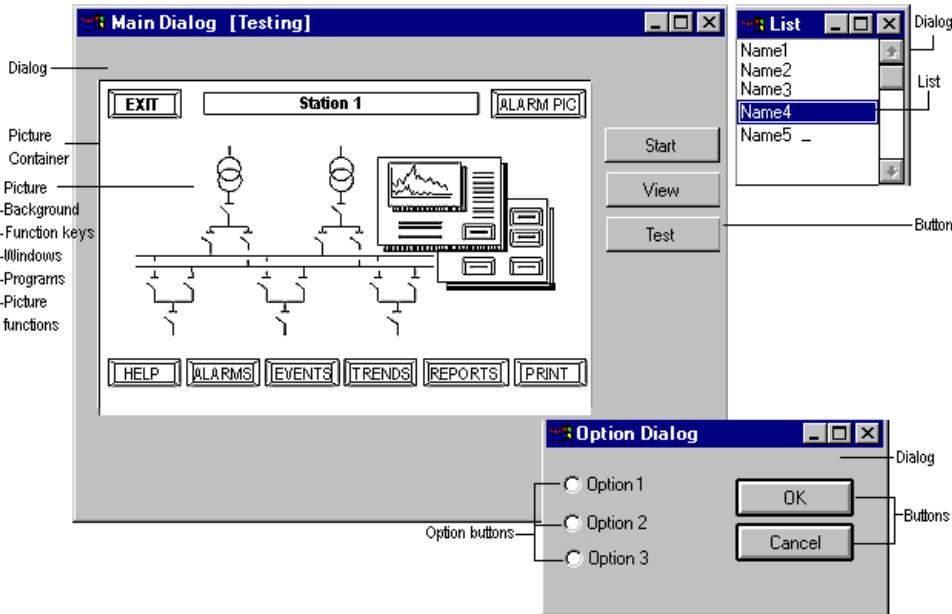


Figure 1: A SYS600 screen view showing three dialogs, one containing a picture.

The application engineering using LIB 500 is discussed in the LIB 500 User's Guide. The user interface design is described in the Visual SCIL User Interface Design and the Picture Editing manuals.

### 3.1.3 What can be done with SCIL

By means of SCIL, the user can control the entire SYS600 system, not only the features related to the application, but also features related to the system configuration and communication. With SCIL, for instance the following actions are possible (as shown in [Figure 2](#)):

- Program the user interface portion of the application, for example, define the dynamic changes in pictures and dialogs, and program operator activated function keys and buttons.
- Design various forms of process control, such as manual control, sequential control, time control, event control, etc.
- Define routines for calculation and updating in databases to be started automatically or manually.
- Design reports for presentation on monitor screens or for paper printout.
- Configure, supervise and handle system components, for example printers and monitors.
- Build process simulations.
- Exchange data with other SYS600 applications and with external (non-SYS600) applications (for example, office applications).

Learning SCIL requires no previous knowledge of conventional programming.

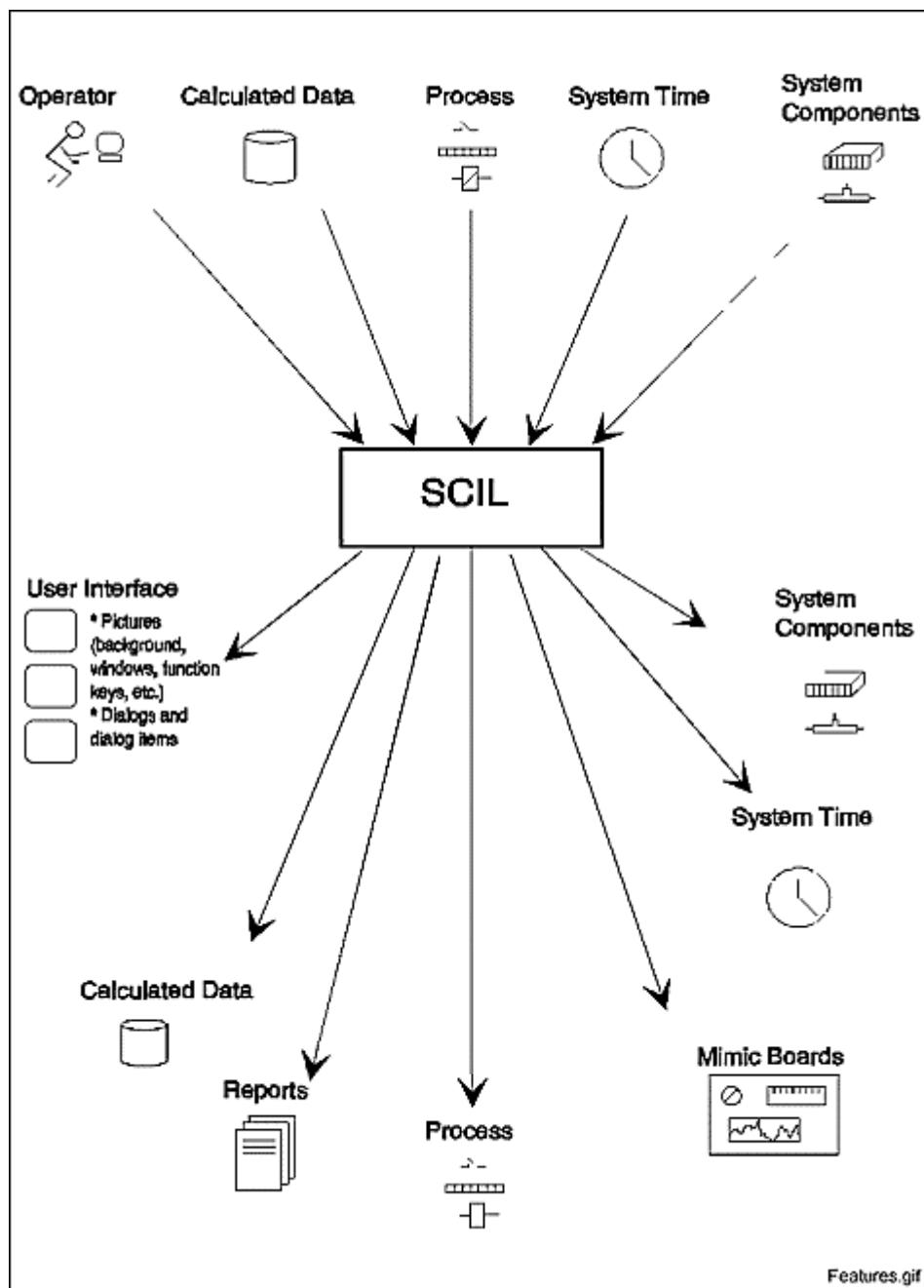


Figure 2: Features controlling the SCIL program execution and features controlled by SCIL

## 3.2 SCIL programs

### 3.2.1 Composition

A SCIL program is composed of one or more textual statements. Each statement represents an instruction to the system about a task to be carried out, for example, the presentation of a picture or the assignment of a variable value. Consecutive statements mean a sequence of instructions, which are carried out in the prescribed order.

### 3.2.2 Occurrence

SCIL programs appear in pictures, in command procedures (objects for automatic or SCIL activated program execution), and in the dialogs and dialog items (Visual SCIL objects). For more information, see [Section 4](#).

### 3.2.3 Activation of SCIL programs

The SCIL programs can be started manually or automatically, for example:

- The function keys in pictures and the buttons in dialogs contain SCIL programs that are started manually by the operator.
- The pictures and dialogs can contain programs, which are started automatically when a picture is entered or exited, periodically with a certain time interval, and on the occurrence of a process event.
- The command procedures can start automatically by real or simulated process events, by time, from other SCIL programs, etc.
- SCIL programs can be started by a SCIL command or a program call.

### 3.2.4 Execution modes

SCIL programs may run in one of two execution modes:

- In Normal Mode, the entire functionality of SCIL language is available for the programmer.
- In Read-only Mode, commands and functions that have side-effects are not allowed. Any operation that alters the data of the base system or devices connected to it is regarded as a side-effect.

The Read-only Mode is used in following cases:

- A classic monitor runs in read-only mode before a successful login and after a logout.
- An unauthenticated OPC client runs as read-only if the system hardening is configured accordingly.
- An external SCIL-API program runs as read-only if the system hardening is configured accordingly.
- Search and query conditions (filters) of database SCIL functions are evaluated in read-only mode.
- SCIL Data Derivation Language (SDDL) expressions are typically evaluated in read-only mode.

### 3.2.5 Example

The example in [Figure 3](#) gives an insight into what a SCIL program can look like. The program could be placed under a function key in a picture, which means that it is executed each time the key is pressed. The purpose of the program is to bring a new picture, called a MENU, on the screen provided that a password is given correctly (999).

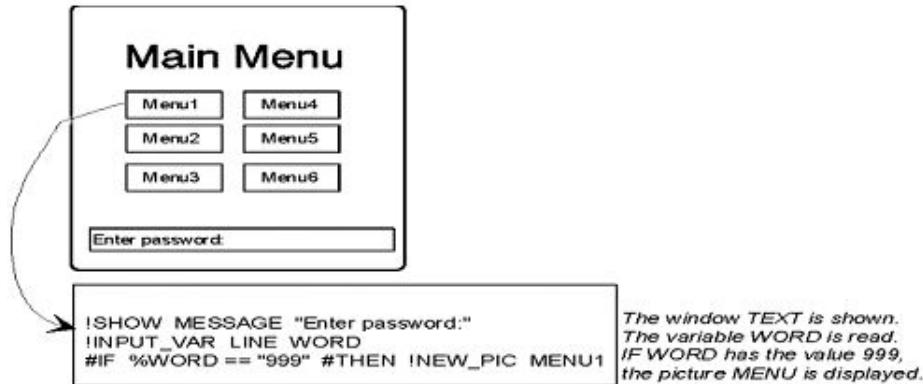


Figure 3: A SCIL program for the input and control of a password

## 3.3 SCIL statements

### 3.3.1 Components

Each line in the program example in [Figure 3](#) is a SCIL statement. A statement can also be continued on several lines. A SCIL statement may consist of the following main components:

- **Commands.** These are words with a pre-declared meaning constituting orders to the system about steps to be taken. In the example in [Figure 3](#), each program line starts with a command.
- **Objects.** An object is a broad concept which may correspond to physical parts of the system or the process, database items, user interface objects, or abstract functions specific to the SYS600 system.
- **Variables.** Variables are facilities for the temporary storage and use of changing data. In the example in [Figure 3](#), WORD is a variable.
- **Function calls.** SCIL has a large number of predefined functions for various purposes.
- **Expressions.** Expressions are formulas, which can contain constants, object notations, variables, function calls and operators (for example, +, -, /, \*).

### 3.3.2 Statement format

The components above can be combined into statements according to one of the following three formats:

command {arguments} (1)

variable = expression (2)

name.program {arguments} (3)

Type (1) (described in [Section 8](#), [Section 10](#) and [Section 11](#)) implies that an action is performed on or by means of the arguments. The arguments may be object references, expressions, variables, names, etc. Some commands do not require any arguments, they are as such complete statements. In some cases, the arguments may contain statements. The command names start with a ! (picture commands), # (control commands) or . (Visual SCIL, full graphics and Motif commands).

Type (2) (described in [Section 6](#)) implies that a variable gets a value.

Type (3) (described in [Section 5](#)) is a program call which starts a program execution in a picture, dialog or dialog item.

## 3.4 Organization of this manual

This manual is composed of 14 sections, which have the following contents:

- |                            |  |
|----------------------------|--|
| <a href="#">Section 3</a>  | This introduction.   |
| <a href="#">Section 4</a>  | Describes the tools for SCIL programming, the rules concerning the structure of SCIL programs, SCIL characters and naming SCIL elements.               |
| <a href="#">Section 5</a>  | Describes the different SCIL data types allowed for data and expressions included in SCIL programs.  |
| <a href="#">Section 6</a>  | Describes briefly the different object types, system objects, application objects, user interface objects and files, and how they are handled in SCIL. |
| <a href="#">Section 7</a>  | Describes the use of variables in SCIL: how to assign values to variables and how to use them, predefined picture variables.                           |
| <a href="#">Section 8</a>  | Describes the construction of SCIL expressions.  |
| <a href="#">Section 9</a>  | Describes the SCIL commands: picture handling commands, control commands and Visual SCIL commands.   |
| <a href="#">Section 10</a> | Describes the predefined SCIL functions which can be included in expressions.  |
| <a href="#">Section 11</a> | Describes the lowest level of the full graphics handling (generally not needed for ordinary application engineering).                                  |
| <a href="#">Section 12</a> | Provides a programmer's quick guide.   |
| <a href="#">Section 13</a> | Describes the SCIL Program Editor.   |
| <a href="#">Section 14</a> | Describes the SCIL Compiler.   |
| <a href="#">Section 15</a> | contains a list of ODBC error codes.   |
| <a href="#">Section 16</a> | describes the contents of parameter files.   |

A list of subject indices can be found at the end of this manual.

# Section 4      Programming in SCIL

This section describes the SCIL programming environment and the rules for programming in SCIL:

- |                             |   |
|-----------------------------|---|
| <a href="#">Section 4.1</a> | <a href="#"><u>SCIL programming environment</u></a> : the SYS600 objects where SCIL programs and expressions are found, the programming tools in brief, the use of SCIL expressions in external applications via DDE. |
| <a href="#">Section 4.2</a> | <a href="#"><u>SCIL programming rules</u></a> : The structure of the SCIL programs, SCIL characters, SCIL names.  |

## 4.1      SCIL programming environment

### 4.1.1    General

SCIL programs appear in:

- Pictures
- Visual SCIL objects
- Command procedures
- Programs communicating with the base system via OPC Data Access Server

In addition, SCIL expressions appear in windows, data objects and time channels. SCIL expressions can also be entered in external Windows based applications and evaluated through DDE (Dynamic Data Exchange).

SCIL programming is carried out online, while the SYS600 system is running. Various application programming tools, such as Dialog Editor, Picture Editor and Object Navigator, use the SCIL Program Editor for entering SCIL programs. The SCIL Program Editor is able to check the syntax of the program. To test SCIL programs, use the **Test** Dialog accessed from the Tool Manager.

### 4.1.2    SCIL program editor

SCIL programs are written in the SCIL Program Editor, which is accessed from tool pictures and from the Tool Manager. The SCIL Program Editor is opened from the Picture Editor, the Dialog Editor and the Command Procedure object definition tool.

The SCIL Program Editor is described in [Section 13](#).

### 4.1.3    Picture programs

A dynamic picture is composed of a static background, windows, function keys, SCIL-programs and picture functions. Windows are the dynamic parts of the picture. They can present data fields, graphs, figures or complete pictures. The picture functions are complete pictures which are integrated in the total picture. Picture functions and window pictures (pictures shown in a window) are commonly called "part pictures" or "sub-pictures", while the total picture is called "main picture".

Pictures are built and programmed in the Picture Editor where they can be named freely. The pictures and picture editing are described in the Picture Editing manual.

A picture may contain the following types of picture programs (none of them is obligatory):

- **A draw program** executed every time the picture is loaded on screen, immediately after the background has been produced on screen but before the start program is executed. It can, for example, be used for adding context specific graphics to the background by means of graphics commands.
- **A start program** executed after the draw program. The start program is used for basic definitions such as initial variable values, update interval, and program blocks.
- **An update program**, which is executed repeatedly at intervals defined by a SCIL command (the !UPDATE command, see [Section 8](#)) as long as the picture is displayed on screen.
- **An exit program**, which is executed each time the picture is closed (even at exit by clicking three times in the upper left corner of the picture).
- **Function key programs** executed at each click on the function keys to which they belong.
- **Named programs** executed by program calls. A SCIL picture may contain any number of named SCIL programs. The named programs are started by program calls as described in [Section 6.4](#). The named program names may be up to 63 characters long. Each picture may contain a named program with a predefined name, ERROR\_HANDLER, where the programmer can define the error handling to be used in the picture. The ERROR\_HANDLER program is described in [Section 12](#).

Besides these programs, the picture contains a background program which is created automatically by the picture editor. The background program contains graphics commands. Normally, it should not be edited manually.

#### 4.1.4 Window definitions

The window definitions, see [Figure 4](#), may contain expressions which specify what is to be shown in the windows. The expression of a window is evaluated each time the window is shown. The window definitions may also contain conditions for the display of different representations. Both the expressions and the conditions follow the rules of SCIL.

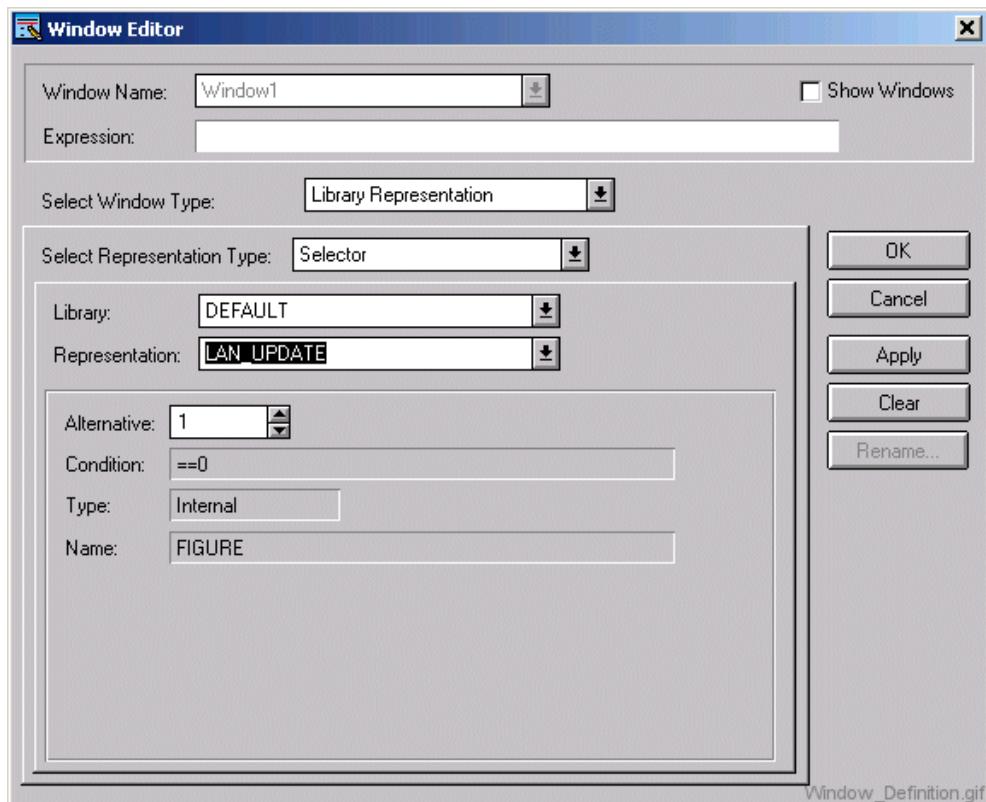


Figure 4: An example of a window definition

## 4.1.5 Methods

Each Visual SCIL object can have a number of methods, most of which are SCIL programs. (There are also predefined methods which are programmed in C, but these methods are not editable).

A dialog or a dialog item may contain the following SCIL programs (methods):

- Methods started at the creation and deletion of the object.
- Cyclically activated methods.
- Event activated methods started by a process event or an event activated by SCIL (through event objects, see [Section 6.3](#)).
- Action methods started on an operator intervention (for example, a click on a button).
- A help method started when help is requested.
- An error handling method.

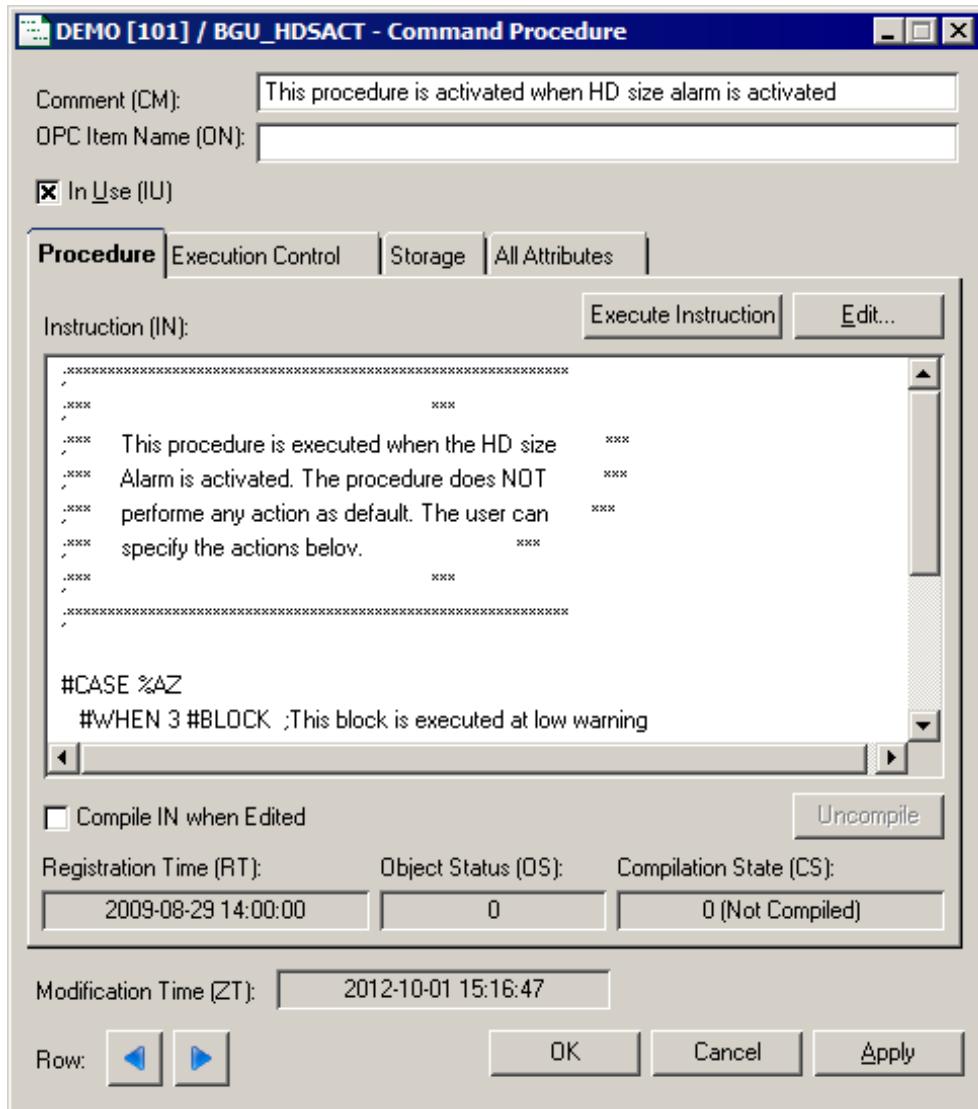
In addition, each Visual SCIL object may have an arbitrary number of user defined methods which are executed by a method call (see [Section 6.4](#)).

The methods of the dialogs and dialog items are programmed in the Dialog Editor. The composition and programming of dialogs is described in the Visual SCIL User Interface Design manual.

## 4.1.6 Command procedures

A command procedure is an independent SCIL program consisting of up to 2 000 000 lines (see [Figure 5](#)). A command procedure can be activated by a time channel (see below), an event channel (controller of event-bound activities) or a SCIL program.

Command procedures are described in the Application Objects manual.



*Figure 5: An example of a command procedure*

## 4.1.7 Data objects

The data objects (datalog objects) are objects for the registration and storage of data. The object definition, see [Figure 6](#), contains a SCIL expression which states how the data is to be calculated.

Data objects are described in the Application Objects manual.

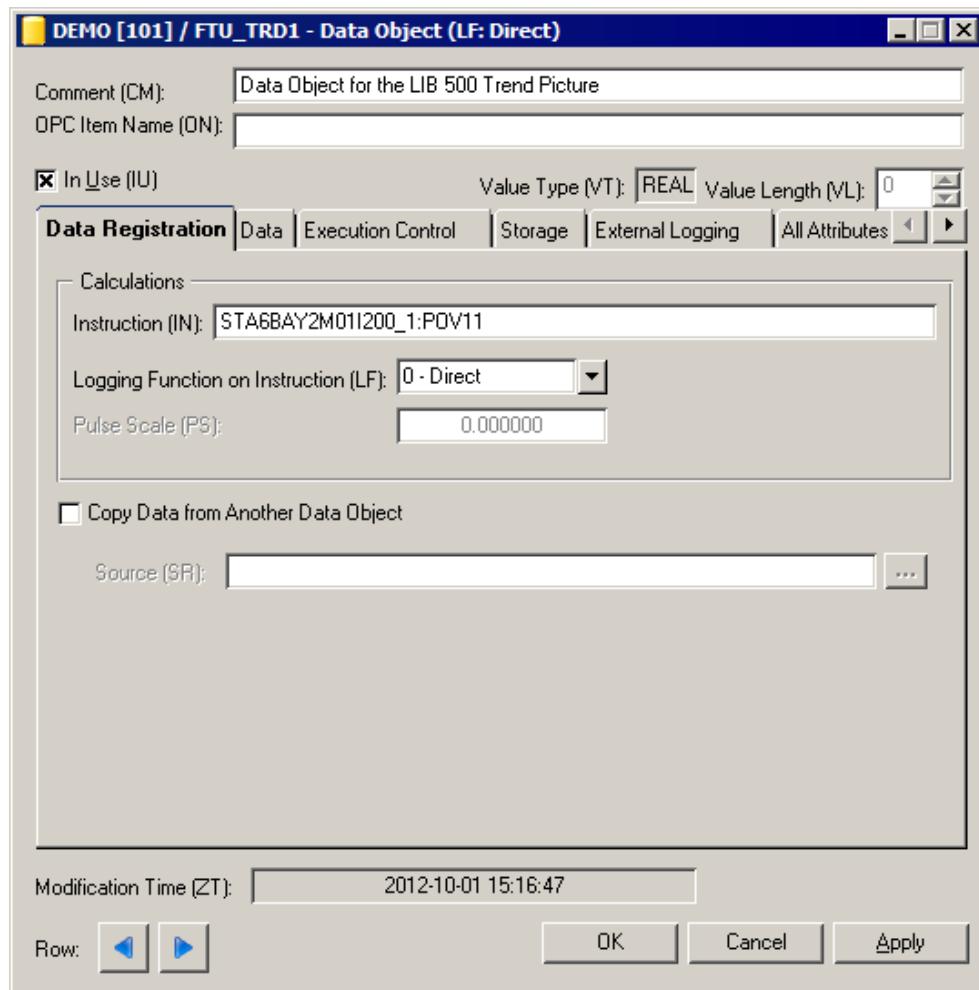


Figure 6: An example of a data object definition

#### 4.1.8 Time channels

The time channels control the execution of time-bound activities. By conditions which are SCIL expressions, see [Figure 7](#), the initialization times and execution times can be restricted, so that an initialization or an execution can only occur when the conditions are fulfilled.

Time channels are described in the Application Objects manual.

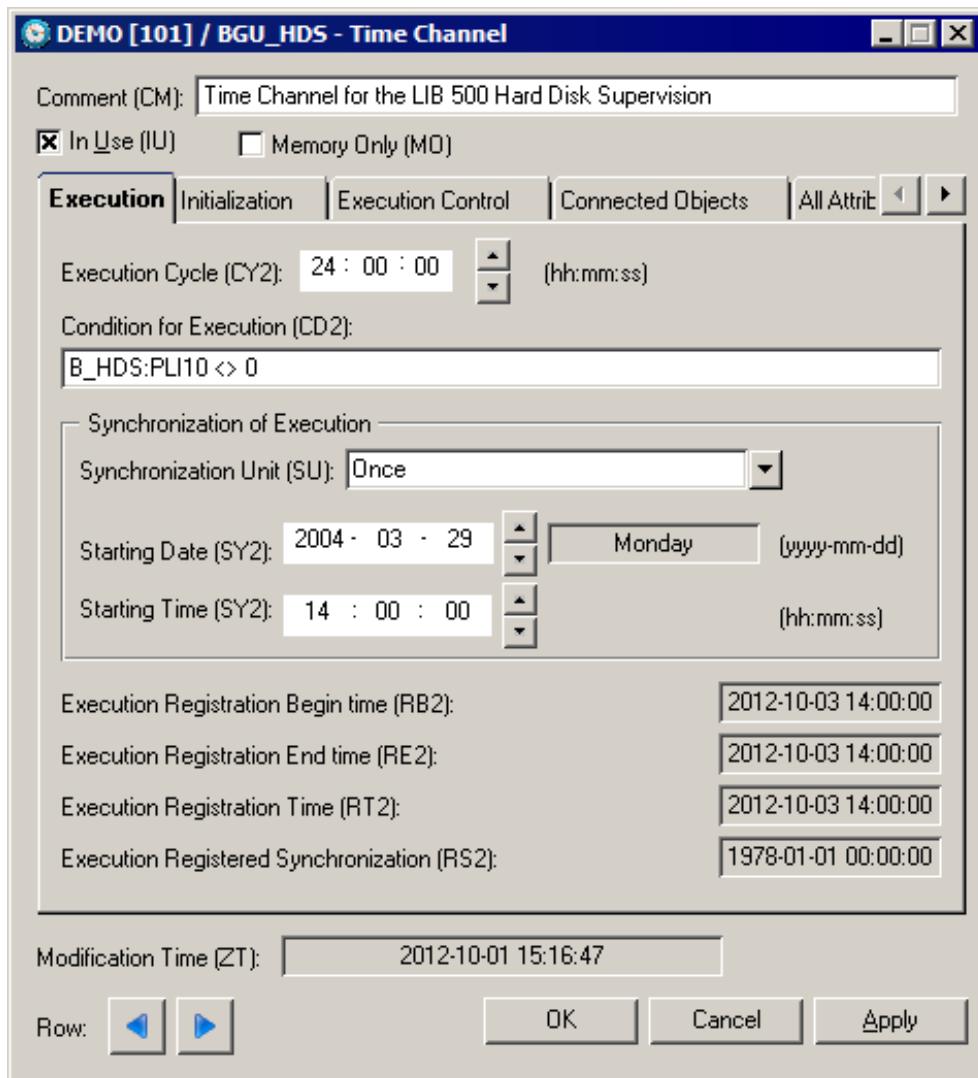


Figure 7: An example of a time channel definition

#### 4.1.9 Using SCIL in external applications

External applications may run SCIL programs and evaluate SCIL expressions using OPC connection to the base system. The item name space of the SYS600 OPC Data Access server contains predefined item names SCIL\_PROGRAM and SCIL used to execute SCIL. See the OPC Server manual for details.

In addition, SYS600 applications can be accessed from external Windows applications using the DDE protocol, so that the SYS600 application works as a DDE server and the other application works as a DDE client.

The DDE identifiers service, topic and item have the following meanings when accessing a SYS600 application:

Service =	MicroSCADA
Topic =	SYS600 application number (APL object number, see the System Configuration manual)
Item, data =	Item and data depends on the DDE transaction as follows:

Table continues on next page

REQUEST:	item = SCIL expression
POKE:	item = SCIL object notation
	data = SCIL expression
EXECUTE:	item = SCIL statement.

The DDE Server function in SYS600 supports the following SCIL data types (see [Section 5](#)): integer, real, text, boolean. There are some SCIL functions that can be used in the composition of the expressions used via DDE. See the DDE Server Functions in [Section 10](#).

When handling data using DDE Server, occurred SCIL errors are preserved so that the user can afterwards read the last SCIL status code of each DDE conversation by doing a request transaction by giving DDE\_SCIL\_STATUS\_CODE as Item value.

Accessing a SYS600 application using the DDE Server requires that the DDE Server has been enabled in the base-system configuration (the SYS:BDE attribute).

For more information about using DDE, refer to the DDE documentation of the Windows Application that is currently in use.

## 4.1.10 SCIL test tool

The Test Dialog accessed from the Tool Manager allows the programmer to enter individual SCIL statements and expressions, as well as programs. It also provides means for detailed examination of the SCIL expressions.

# 4.2 SCIL programming rules

## 4.2.1 Program structure

A SCIL program can contain up to 2 000 000 lines, and each line up to 65 535 characters (including spaces). A SCIL statement comprises one or more lines. A minus sign (-) at the end of a SCIL line, before a possible comment, indicates that the statement continues on the next line. A line may be divided anywhere where spaces are allowed, but not within text constants. Empty lines are allowed anywhere in the programs.

Spaces are allowed anywhere in the program except within numbers, words, names, object notations and composed symbols. These elements, disregarding operators, must be adjacent to at least one space at each side. Also the operators DIV and MOD (see [Section 8](#)) must be adjacent to spaces.

Upper and lower case letters may be freely intermixed. When the program is executed, lower case letters are converted into upper case, except for lower case letters within text constants (see [Section 5](#)).

Comments can be placed anywhere in the program. They are preceded by a semicolon (;), which indicates that the rest of the line is a comment. However, if the semicolon is enclosed in quotation marks (";"), it is regarded as a text (see [Section 5](#)).

## 4.2.2 Examples

The following two SCIL-programs are functionally equivalent:

Example 1:

```

T = TEMP:PAI1
#IF T > 90 #THEN #BLOCK
    #SET C:PBO2 = 0
    #PRINT 2 OVERHEAT
#BLOCK_END
#ELSE_IF T < 70 #THEN #SET C:PBO2 = 1

```

Example 2:

```

T = TEMP:PAI1 ;READ THE TEMPERATURE
#IF T > 90 #THEN #BLOCK ;IF WARMED OVER 90
    #SET C:PBO2 = 0 ;STOP HEATING
    #PRINT 2 OVERHEAT ;PRINT OVERHEAT MESSAGE
#BLOCK_END
#ELSE_IF T < 70 #THEN #SET C:PBO2 = 1
                                ;RESTART HEATING IF T < 70

```

The following statement is divided on two lines. As a text constant cannot be divided it is necessary to type it as a sum of two texts.

```

A = "IN THIS EXAMPLE, A LONG TEXT VALUE IS " - ;Comments allowed
      + "ASSIGNED TO VARIABLE A" ;like this

```

### 4.2.3 SCIL characters

SCIL uses Unicode character set to encode text data. All current languages may be written using 2-byte Unicode characters in numeric range 0 ... 65535. The range also contains a lot of special purpose 'characters', such as mathematical symbols and many others.

However, the Unicode standard allows for representation of more than 1 million 'characters'. The characters outside the range 0 ... 65535 are used to write ancient languages and to draw various additional symbols. These characters occupy 4 bytes and may be embedded in 2-byte character strings. These 4-byte Unicode characters are allowed in SCIL text strings, but they are counted as 2 characters wherever the length of string is concerned.

Some of the characters and character combinations have a special meaning in SCIL. They symbolize punctuation marks and operators or have a special meaning in the SCIL Language. These special symbols and their meanings are listed below. When the symbols are composed of more than one character, the characters can not be separated by spaces.

Symbol	Meaning in SCIL
-	minus sign, continuation of program line
+	plus sign
*	multiplication
**	exponential operator
/	division, separator in paths
\	separator in Visual SCIL object paths
(	parentheses, enclose indexing and argument lists
,	an enumeration
.	decimal point, marks a graphics or Motif command, an attribute or a method
..	index range

Table continues on next page

Symbol	Meaning in SCIL
"	encloses a text
@	global variable assignment
%	global variable access
'	variable expansion
#	precedes a control command
!	precedes a picture command
:	follows an object name
;	starts a comment
=	assignment
==	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<>	not equal to
^	an octal number

#### 4.2.4 SCIL names

Most user interface objects (picture and Visual SCIL objects, see [Section 6](#)) and application objects (see [Section 6](#)) are identified by logical names (identifiers), which can be chosen freely. Likewise, the names of variables, named programs, user defined methods, logical library representation files and directory paths can be named freely.

As a rule, the names may be up to 63 characters long. However, the names of pictures, windows and logical paths may be only 10 characters long.

Allowed characters are the letters A-Z, all digits and underscore (\_). In application object names periods (.) are allowed as well. As a rule, the first character of a name must be a letter or an underscore. Application object names may also start with a digit, but this is not recommended.

SCIL supports blanks in Windows directory and file names.



Upper case and lower case letters are not distinct. For example, OBJ\_NAME and Obj\_name are the one and the same name.

#### Examples

Below are some examples on valid and invalid SCIL names:

Valid names:

RELAY

RELAY\_STN

RELAY\_1234

Invalid names:

Table continues on next page

4TH_RELAY	(starts with a digit, not valid as a name of a picture, window or variable)
RELÄ	(contains a special character)
RELAY_STATION	(too long for a window name)

# Section 5 Data types

This section describes the SCIL data types: integer, real, boolean, time, text, bit string, byte string, vector and list.

## 5.1 General

### 5.1.1 Data types

In SCIL, there are six types of simple data (data constituted of single values): integer, real, boolean, time, text, bit string and byte string.

In addition, there are two types of structured data: the data type vector that represents an ordered array of data elements, and the data type list that represents an unordered list of attribute names and attribute values. The components (elements and attributes) of structured data may be of any data type, for example an element of a vector may be a vector or a list.

Data type none is used in some contexts to denote a missing value.

For generic functions that deal with any type data (for example read the type or status of data, determine the length of data item or test two data items for equality), see [Section 10.2](#).

### 5.1.2 Reliability of data

In SCIL, each data item has a status code, which expresses the reliability of the data. As a rule, when an expression is evaluated, it gets the status code of the least reliable operand. Data written explicitly by the programmer or entered by the user always have OK status. A bad status code may originate, for example, from a process object value that has been marked as obsolete due to a communication fault.

The status code of data can be read and written with SCIL functions GET\_STATUS and SET\_STATUS (See [Section 10.2](#)). The status codes are listed in the Status Codes manual.

## 5.2 Integer

### 5.2.1 Description

The integer data type denotes positive and negative integer values ranging from -2 147 483 648 through to +2 147 483 647. These values may be referred to by symbolic names MIN\_INTEGER and MAX\_INTEGER, respectively. Constant integers outside these limits are represented as real numbers. The most negative integer -2 147 483 648 cannot be written as a constant, it should be referred to by its symbolic name MIN\_INTEGER.

Integer constants can be typed either in decimal or in octal form. In decimal form integers are written as a sequence of digits 0 ... 9 optionally preceded by a sign. No embedded spaces nor commas or points are allowed.

Octal constants are written as a sequence of octal digits 0 ... 7 followed by a trailing up arrow (^).

## 5.2.2 Example

Examples of some valid integer constants:

256

0

000123

-5

777^

The last one represents integer value 511(decimal).

## 5.3 Real

### 5.3.1 Description

The real data type expresses, with limited precision, the continuous quantities within a large range of values. The accuracy of real numbers is limited to about seven significant decimal digits.

[Table 1](#) shows the value ranges of the real numbers. All numbers between the smallest negative and the smallest positive numbers yield the value 0. The ranges are determined by the processor of the base system computer.

*Table 1: The value ranges of real values.*

Smallest negative value	-5.88 E(-39)
Largest negative value	-3.41 E38
Smallest positive value	5.88 E(-39)
Largest positive value	3.41 E38

Real numbers are written using digits, decimal points and signs. No exponents are allowed. At least one digit must precede the decimal point. Numbers without any decimal point are considered as integer values if they fall within the value range of integer values. Outside that range, they are considered real values.

Internally, real numbers are represented by a 32-bit floating point format.

Because the internal representation of a real number is generally not exact, care should be taken when comparing two real values for equality. For example, it is not safe to rely on such facts as  $1.7/5.0 == 0.34$ . However, the following holds: every whole number in the range -16 777 216 .. +16 777 216 has an exact representation as a real number. Therefore, for example the relation  $170.0/5.0 == 34.0$  is safe, no rounding errors can occur.

For various arithmetic functions that operate on real data, see [Section 10.3](#).

If better accuracy of floating point arithmetics is required by an application, special purpose SCIL functions operating on 64-bit floating point numbers may be used. See [Section 10.3](#) for functions whose name is prefixed by HIGH\_PRECISION\_.

## 5.3.2 Example

Examples of some valid real numbers:

0.0

1234.56789

2147483648

5.

0.00000000000000000000000001

The second number contains 9 significant digits and thus cannot be stored without some loss of precision. The third number is a real constant because it falls outside the range of integers.

## 5.4 Boolean

### 5.4.1 Description

Boolean data may take two values: FALSE and TRUE.

Boolean values result from comparisons (relational operators) and various other tests (such as functions EVEN and ODD). They are frequently used in conditional statements and expressions.

Internally, FALSE is represented by the value 0 and TRUE by the value 1. Hence, the relation between the two boolean values is:

FALSE < TRUE.

Integer 0 and 1 are returned when boolean SCIL expressions are used via DDE (Dynamic Data Exchange).

### 5.4.2 Examples

Some examples of boolean expressions and their values:

Expression	Boolean value
4 + 1 == 5	TRUE
"NIGHT" == "DAY"	FALSE
ODD(5)	TRUE

## 5.5 Time

Time data is obtained by reading the system clock and various time stamp attributes of objects.

A time value is internally represented as a 32-bit unsigned integer counting seconds since the beginning of 1978.

For various functions that operate on time data, for example give a textual representation (for example, calendar time) of time data, see [Section 10.4](#).

## 5.6 Text

### 5.6.1 Description

Text data is composed of 0 to 65 535 characters. Unicode character set encoding is used. Symbolic name MAX\_TEXT\_LENGTH may be used to refer to the text length limit of the current implementation (65 535 today). Strictly speaking, the limit 65 535 refers to the maximum length of the text string as 2-byte words. The Unicode standard defines also characters that occupy 4 bytes of memory. Fortunately, they are very seldom used. Hence, for practical purposes it is safe to equate the length of text string and the number of characters it contains.

Text values may be compared and concatenated (added using + operator), see [Section 8](#). For various predefined functions that operate on text data, see [Section 10.5](#).

Text constants are enclosed in quotation marks (""). A single quote ('') or double quote (") contained in the text must be typed as two single quotes or two double quotes respectively.

### 5.6.2 Examples

Two text constants:

"ABC\_123!!?%"

"This is a text constant containing one "" and one ''"

## 5.7 Bit string

### 5.7.1 Description

A bit string is a sequence of bits stored in consecutive memory bytes. The length of a bit string (the number of bits in the string) may be anything from 0 to 65 535. The bits in a string are numbered from 1 to 65 535 counting from left to right. Symbolic name MAX\_BIT\_STRING\_LENGTH may be used to refer to the bit string length limit of the current implementation (65 535 today).

Bit strings appear as values of process object attributes and functions. They can be created using functions BIT\_SCAN and BIT\_STRING and represented in a textual format by means of BIN function (See [Section 10](#)).

Bit strings may be compared and concatenated ('added' using + operator), see [Section 8](#). For functions that manipulate bit strings on bit level, see [Section 10](#).

Internally, a bit string is represented as a two-byte length field followed by as many data bytes as needed to store the bits.

### 5.7.2 Example

An example of a bit string of eight bits converted to a text by means of the BIN function:

BIN(%BITSTRING) == "01010101"

## 5.8 Byte string

### 5.8.1 Description

A byte string is a sequence of consecutive 8-bit bytes with no structure and no predefined semantics. The length of a byte string may be anything between 0 and 8 388 600 bytes. Symbolic name MAX\_BYTE\_STRING\_LENGTH may be used to refer to the byte string length limit of the current implementation (8 388 600 today).

Byte strings can be created by the PACK\_STR function. By using the function UNPACK\_STR, a byte string can be interpreted as an array of numerical values. Byte strings may be used to exchange binary data between SYS600 and other applications. They are used within the SCIL programming environment, for example, to store compiled SCIL programs.

Byte strings may be compared and concatenated (added using + operator), see [Section 8](#).

Internally, a byte string is represented as a four-byte length field followed by as many data bytes as needed to store the bytes.

## 5.9 Vector

### 5.9.1 Description

A vector is an ordered array of data. A data item as a component of a vector is called an element. A vector may contain up to 2 000 000 elements numbered from 1 to 2 000 000. Symbolic name MAX\_VECTOR\_LENGTH may be used to refer to the vector length limit of the current implementation (2 000 000 today).

The elements of a vector may be of any data type. Different elements may even be of different data types.

A vector may be expanded by simply assigning values to its elements or by function INSERT\_ELEMENT. Elements may be removed by function DELETE\_ELEMENT. For these and various other functions that operate on vector data, see [Section 10.7](#).

For accessing elements of a vector, see [Section 5.11](#).

### 5.9.2 Vector aggregate

A vector can be written as a vector aggregate in the following format:

`VECTOR([element [,element]*])`

or

`(element1, element2 [,element]*)`

The key word VECTOR is followed by the vector element values separated by commas and enclosed in parentheses. The elements may be given as expressions of any data type. The key word VECTOR may be omitted if two or more elements are listed. An empty vector may be written as VECTOR or VECTOR().

### 5.9.3 Example

An example of a vector aggregate of five elements:

(-23, "NAME", 0.000001, CLOCK, A + B)

## 5.10 List

### 5.10.1 Description

A list is an unordered collection of data components. The components are called attributes which have a name and a value. Hence, data type list is a list of attribute name/value pairs. A list can hold up to 2 000 000 attributes. Symbolic name MAX\_LIST\_ATTRIBUTE\_COUNT may be used to refer to the attribute count limit of the current implementation (2 000 000 today).

Attribute names are freely chosen, up to 63 character long identifiers.

The attributes of a list may be of any data type. Different attributes may have different types.

New attributes to a list may be added by function MERGE\_ATTRIBUTES. Function DELETE\_ATTRIBUTE may be used to remove attributes. For these and other functions that operate on list data, see [Section 10.8](#).

For accessing attributes of a list, see [Section 5.11](#).

### 5.10.2 List aggregate

A list can be written as a list aggregate in the following format:

LIST([(attribute = value [,attribute = value]\*)])

The key word LIST is followed by attribute value assignments separated by commas and enclosed in parentheses. The attribute values may be given as expressions of any data type. An empty list may be written as LIST or LIST().

### 5.10.3 Example

A list could have the following contents:

Attribute name	LN	IX	UN	OA	OB	OV	....
Index							
1	"A"	4	10	3560	7	0	....
2	"B"	2	11	3430	16	5.5	....

Using a LIST aggregate and vector aggregates, the list could be written:

LIST(LN = ("A","B"), IX = (4,2), UN = (10,11), OA = (3560,3430), OB = (7,16), OV = (0,5.5))

## 5.11 Accessing components of structured data

Components of structured data, that is, elements of a vector and attributes of a list, are accessed by the following notation:

entity[component]\*

where each component is one of the following:

( index )

( [low] .. [high] )

.attribute

'index'	Integer value, 1 ... 2 000 000. Selects an element of a vector.
'low', 'high'	Index range, 'low' and 'high' integer values 1 ... 2 000 000. Selects a subvector, or a slice, containing the elements 'low' to 'high' from a vector, 'high' must be greater or equal to 'low'. If 'high' is omitted, all indices from 'low' to the end of vector are selected. If 'low' is omitted, all indices up to 'high' are selected, that is, 'low' defaults to 1.
'attribute'	Name of an attribute of a list.

An index range may be specified only as the last component of the notation.

There is no limit for the number of components in the notation. Consequently, any component of structured data may be directly accessed, regardless of the complexity of the data structure.

'entity' is one of the following:

- Name of a local variable or an argument (see [Section 7](#)).
- Name of a global variable preceded by % or @ (see [Section 7](#)).
- Visual SCIL object attribute reference (see [Section 6](#)).
- Window attribute reference (see [Section 6](#)).

A component access notation may appear as an operand of an expression and as the left hand operand of an assignment statement and .SET command.

Assigning a value to an attribute does not create the attribute. If the attribute to be assigned does not exist, error SCIL\_ATTRIBUTE\_NOT\_FOUND is raised.

On the other hand, assigning a value to a vector element that does not exist expands the vector or even creates the vector, if the target is not of vector type.

Examples:

Valid statements containing component access notations:

A.B = C(2).D(2 .. 6)	;Local variable access
@A.B(1)(12) = %C(2).D	;Global variable access
.SET .HEIGHTS(5) = ROOT\OBJ._GEOMETRY.H	;VS object attribute access
.SET ROOT/WINDOW1.A.B = WINDOW2.C(1)	;Window attribute access

Invalid statements:

#SET A:VB.C = SYS:BUV(5).A	;Not for application objects
MS = LOCAL_TIME.MS	;Not for return values
A = .METHOD(ARG).A	;Not for return values
(VECTOR1 + VECTOR2)(1)	;Not for intermediate results



# Section 6 Objects and files

This section describes in general terms the objects and how to handle them in SCIL. The objects are of three main categories which are described in the following sections:

<a href="#">Section 6.1</a>	<b>General:</b> An overview of the three object categories, some object related concepts.
<a href="#">Section 6.2</a>	<b>System objects:</b> The communication system objects and base system objects, their object notation and some attributes.
<a href="#">Section 6.3</a>	<b>Application objects:</b> The application object types, their object notation, and some attributes.
<a href="#">Section 6.4</a>	<b>User interface objects:</b> The pictures and picture components, the visual SCIL objects, the object hierarchy, the object references, attributes and methods.
<a href="#">Section 6.5</a>	<b>Files:</b> File naming and different file types.

The description of the object types here is superficial. For a detailed description, refer to the following manuals: System Objects, Application Objects, Visual SCIL Objects, Visual SCIL User Interface Design and Picture Editing.

## 6.1 General

### 6.1.1 Object categories

There are three categories of objects which represent three levels of system engineering:

- **System objects.**  
System objects are used for system configuration and communication. There are two types of system objects:
  - Base system objects.
  - Communication system objects (previously named system objects).
- **Application objects.**  
These objects form the functional portion of the applications. There are eleven types of application objects:
  - Process objects
  - Event handling objects
  - Scale objects
  - Data objects
  - Command procedures
  - Time channels
  - Event channels
  - Logging profiles
  - Event objects
  - Free type objects
  - Variable objects
- **User interface objects.**  
There are two types of objects for composing the user interface portion of applications:
  - Pictures
  - Visual SCIL objects. There are approximately 40 Visual SCIL object types. Each type corresponds to a type of dialog, dialog item (for example, buttons, texts, lists, menus) or image.

## 6.1.2 Attributes and methods

An important concept when talking about objects is the notion of attribute. Most object types have attributes which represent the values and properties of the objects. The value of a process object, for example, is represented by an attribute, the time stamp is another attribute, and so is the alarm state. Regarding the Visual SCIL objects, the attributes may correspond to visual properties such as the label or color of a button.

Through the attributes, the SCIL programmer can use and change the object properties. For instance, the color of a dialog item and the alarm limits of a process object may be changed by means of a SCIL statement.

The Visual SCIL objects may also have a number of methods which are programs written in SCIL or in C. The SCIL programs may be freely modified, while the C programs are predefined. By executing predefined methods, the programmer can affect certain features of the objects.

## 6.1.3 Handling objects in SCIL

The values and properties of the objects can be read in SCIL as attributes. Provided that write access is allowed, the values and properties can also be written by means of SCIL commands. Reading an attribute means that it is used in a SCIL expression, for example, assigned to a variable or shown in a window.

The system objects and application objects, more precisely their attributes, are accessed by means of an object notation which contains the object name and type and the attribute name. The Visual SCIL objects are accessed by means of an object name or path and an attribute name.

The system objects and most of the application objects are global and accessible not only within the same application and the same base system, but also in the entire distributed SYS600 system. The Visual SCIL objects, on the contrary, are accessible from SCIL only within the same dialog system.

# 6.2 System objects

## 6.2.1 General

### 6.2.1.1 Overview

The system objects define the hardware and software configuration of the entire SYS600 system, as well as the data communication with connected devices. There are two types of system objects:

- **Base system objects (B)** which define the base system configuration.
- **Communication system objects (S)** which are images of the physical system devices connected to the communication units (NETs).

The base system objects are stored in the RAM memory of the base system computer as long as SYS600 is running. They are not stored on disk, but must be defined at each start-up of the base system. The communication system objects are stored in the communication units (NETs) as long as the units are running. Default values may be stored in the communication programs as a preconfiguration.

### 6.2.1.2 System object notation

SCIL refers to a system object value, i.e. an attribute, by an object notation of the following format (the items within brackets may be omitted):

**name:[application]type attribute[index]**

where

name	is the name of the object. The system objects have predefined names which are composed of a three-letter type notation and a sequence number. Base system objects may also have a user-defined name.
application	is an application number. Generally not needed.
type	is a character indicating the object type: S = communication system object, B = base system object.
attribute	is the attribute to be read or written with the notation. The attribute names are a combination of two letters (A ... Z). The attribute determines the data type of the entire notation. The system object attributes are all detailed in the System Objects manual.
index	is a number or a range of numbers. The indices have different meanings depending on the object type and attribute. For instance, for NET line attributes the index generally refers to the NET line number. For some attributes, the index denotes an address. For more detailed information about the indexing of attributes, see the System Objects manual.

An index or index range is given in any of the following ways:

- As an integer constant, 0 ... 2 000 000, either a positive decimal number or an octal number. Octal numbers are used when addressing bits of memory attributes of communication system objects.
- As an integer type expression enclosed by parentheses.
- As a range enclosed in parentheses, (i .. j), where 'i' denotes the first index number and 'j' the last. Two points surrounded by parentheses, (..), denotes all the indices of the actual object notation. (i ..) means all indices larger than or equal to 'i', and (.. j) all indices less than or equal to 'j'. In an index range, either both or none of the index limits may be given with bit addresses.

There may be no spaces between the items in the object notation.

### 6.2.2 Base system objects (B)

#### 6.2.2.1 Description

The base system objects correspond to the devices and applications located in, connected to, or otherwise known to the base system. They determine the hardware and software properties of the base system and its applications.

The base system objects have the following names:

SYS	The base system itself
APLn	Applications (n = 1 ... 250)
PRIn	Printers ( n = 1 ... 20)
MONn	Monitors (n = 1 ... 100)
STAn	Stations (n = 1 ... 50 000)

Table continues on next page

STYn	Station types (n = 1 ... 31)
NODn	Nodes: base systems and NETs (n = 1 ... 250)
LINn	Links: connection line (n = 1 ... 20)

'n' represents an ordinal number (base system object number). For APL and MON type objects, it can be omitted from the object notations, which means that the notation refers to the current application or monitor respectively.

### 6.2.2.2 Some attributes

Each of the base system object types have their own attributes. Here are some examples:

AS	Application State
	The state of the application (APL): HOT = active, WARM = not active, but accessible, COLD = passive, not available.
LP	Lines per Page
	Number of lines per printed page. Belongs to PRI objects.
AC	Alarm Count
	Belongs to APL objects and shows the number of active alarms in the application. The alarm class is given as an index. Index 0 refers to the total number of active alarms.

### 6.2.2.3 Examples

Examples of some base system object notations:

Notation	Value	Explanation
APL1:BAS	"HOT"	Application number 1 is active.
PRI1:BDT	"NORMAL"	Printer number 1 is black and white.
PRI2:BLP	70	Printer number 2 writes 70 lines per page.
APL1:BAC	20	There are 20 active alarms in application 1.

## 6.2.3 Communication system objects

### 6.2.3.1 Description

The communication system objects correspond to the devices connected to the communication units. Hence, these devices - communication units, stations, other base systems, workstations and peripherals - can be accessed and controlled from SCIL as communication system objects. Each NET unit has its own set-up of communication system objects.

The communication system objects have the following names:

NETn (or NODn)	Communication units and base systems (n = 1 ... 250)
APLn	Applications (n = 1 ... 250)
STAn	Stations (RTUs, PCLs, relays, etc.) (n = 0 ... 255)
PRIn	Printers (n = 1 ... 8)

The 'n' above indicates the number of the objects as known to the NET where they are defined. When handling the objects in SCIL, the 'n' in the object name is the logical number of the device (according to the device mapping attributes).

### 6.2.3.2 Some attributes

Each of the object types have their own attributes. Here are some examples:

SA	Station Address	
	The station address of a device. The NET objects and STA objects have a station address.	
IU	In Use	
	States whether or not the object is in use. Most system objects have this attribute.	
	Value:      0 = out of use 1 = in use.	
PO	Protocol	
	The protocol of a communication line given as an integer, for example, 1 = ANSI full duplex, 14 = SPA. 0 = the line is not defined.	
ME	Memory address	
	The contents of the memory address given as an index. Belongs to the STA objects (ANSI stations).	

### 6.2.3.3 Examples

Examples of some communication system object notations:

Notation	Value	Explanation
STA1:SSA	201	Station 1 has station address 201.
NET1:SPO2	14	Line 2 of NET1 uses the SPA protocol.
PRI4:SIU	1	The printer is in use.
STA5:SME1234^5	0	The contents of storage address 1234, bit number 5 in STA5

## 6.3 Application objects

### 6.3.1 General

#### 6.3.1.1 Object types

The application objects are programmable units which perform various functional tasks in the SYS600 application. They constitute data images of physical process or system devices, data registers, SCIL programs, scaling algorithms, facilities for automatic activation, etc.

There are eleven types of application objects:

- **Process objects (P):** data images of the physical process devices connected to process stations (remote terminal units (RTUs), protective equipment, PLCs, etc.).
- **Event handling objects (H):** texts related to object states and events.
- **Scales (X):** algorithms for scaling analog process values.

- **Data objects (D):** collections of stored data.
- **Command procedures (C):** SCIL programs.
- **Time channels (T):** facilities for automatic time bound activation.
- **Event channels (A):** facilities for automatic event bound activation.
- **Logging profiles (G):** definitions for logging data into Historian database(s).
- **Event objects (E):** mechanisms for event activated start-up of SCIL programs or program sequences in pictures and dialog objects.
- **Free type objects (F):** special objects for the definition of user-defined process object types.
- **Variable objects (V):** temporary objects which can contain attributes collected from other objects or arbitrary attributes.

Process objects, event handling objects, scales and free type objects are stored in the process database, which holds an image of the process. Data objects, command procedures, time channels, event channels and logging profiles are all stored in the report database, which is a database supporting reporting, calculation and control. These objects are also commonly named reporting objects. Variable objects are stored in the same way as variables (see [Section 7](#)). Event objects are not stored at all.

All application objects, with the exception of the variable objects, are global and accessible throughout the entire system.

### 6.3.1.2 Application object notation

SCIL refers to an object value, i.e. an attribute, by an object notation of the following format (the terms in brackets may be omitted in cases where they are not needed):

name:[application]type[attribute][index]

where

name	is the object name. The application object names are logical (symbolical) and can be freely chosen in accordance with the rules given in <a href="#">Section 4.2</a> .
application	is the logical number of the application where the object is stored. Application number must be given if the object belongs to an application other than the current one (the application where the notation is used). If the object belongs to the same application, the number is omitted. The application numbers are defined by the APL:BAP attribute (see <a href="#">Section 7</a> ).
type	is the type of the object given by using the letters P, H, X, D, C, T, A, G, E, F or V.
attribute	is the attribute to be read or written by the notation. As a rule, the attribute names are a combination of two letters (A ... Z). However, if the object is a variable object, the attribute name may be composed of up to 63 characters. The object notation may contain one or no attribute name. The attribute determines the data type of the entire notation when used in expressions.
index	is a number or a range of numbers used to select an element or a range of elements of a vector attribute, or distinguish between process objects within an object group.

As a rule, the indices refer to the elements of an attribute of vector type. The predefined process object types are an exception. For these objects, the indices refer to the individual objects in a group, not to attributes of vector type. Still, for a certain attribute, the values are handled as elements in a vector.

An index or index range is written in one of the following ways:

- As an integer constant, 1 ... 2 000 000. This way may be used only when the length of the attribute name is exactly 2 characters.
- As an integer type expression enclosed in parentheses.
- As a range (i .. j), where 'i' denotes the first index and 'j' the last. Two points surrounded by parentheses, (..), is interpreted as all the indices of the attribute. (i ..) means all indices larger than or equal to 'i', and (.. j) all indices less than or equal to 'j'.

No spaces are allowed between the items in the object notation.

## 6.3.2 Process objects (P)

### 6.3.2.1 Description

The process objects correspond to physical devices connected to the process stations (RTUs, protection equipment, PLCs), for instance switches, sensors and breakers. Each input and output connection in the stations is represented by a process object. Normally, the value of an input object is updated from the process station, while the value of an output object is sent to the station when written with the SCIL command #SET ([Section 9.2](#)).

Some process objects have no correspondence in the stations. These are called fictitious process objects and they can generally be updated only from SCIL programs. They can be used, for example, for process simulation or manually updated data.

There are nine predefined process object types depending on the type of object value (i.e. the input or output connection in the stations): binary input and output, analog input and output, digital input and output, double binary indications, pulse counters and bit streams. In addition, for special purposes, the programmer can define their own process object types by means of free type objects.

An updating of a process object value may cause an alarm, an automatic printout, an updating on screen (through event objects, [Section 6.3.10](#)), and activation of an event channel ([Section 6.3.8](#)).

Process objects with active alarms are included in an alarm buffer, which can be read by SCIL function APPLICATION\_ALARM\_LIST and displayed as alarm lists. If desired, the process object events can be stored in a history database, which can similarly be read by SCIL function HISTORY\_DATABASE\_MANAGER and displayed as event lists.

### 6.3.2.2 Some process object attributes

A few process object attributes are described below. All attributes are detailed in the Application Objects manual.

OV

Object value

The value of the process object as registered in the process database. The object value is the value read from the process (input values) or sent to the process (output values). The attribute is a common name for the attributes BI, BO, AI, AO, DI, DO, DB, PC and BS (see below). Each object value has a time stamp and an error status code.

Value: 1 or 0 for binary objects, real values for analog objects, bit string for bit streams and integer for the others.

BI

Binary Input

Binary process value from the process to the control system.

Value: 1 or 0.

BO	Binary Output
	Binary set value from the control system to the process.
	Value: 1 or 0.
AI	Analog Input
	Analog measurement value from the process to the control system.
	Value: real.
AO	Analog Output
	Analog set value from the control system to the process.
	Value: real.
AL	Alarm
	States whether the object is in alarm state or not.
	Value: 0 - no alarm 1 - alarm state.
HI, LI	Lower Input, Higher Input
	Lower and higher alarm limits. These attributes are defined only for analog objects with an alarm function.
	Value: real.

If object notation is given without any attribute, the attribute is assumed to be OV, except in connection with the commands #LIST, #CREATE, #DELETE and #MODIFY (see [Section 9](#)), where the whole object is referenced to.

### 6.3.2.3 Groups and indices

Related process objects (up to 65 535) of the predefined types may be given the same name. Process objects with the same name form a process object group. The individual objects in a group are accessed by means of indices. A process object notation of any predefined type without an index is interpreted as the lowest indexed object with the given name and attribute (attributes common to the group are always used without indices).

Concerning user-defined types (defined by free type objects), the indices refer to the elements of a vector type attribute. A notation without indexing means the entire vector.

### 6.3.2.4 Examples

Some examples of process object notations:

Notation	Value	Explanation
SWITCH:P	1	The value of the process object (the lowest indexed object in the group).
SWITCH:POV	1	- " -
SWITCH:PBI23	1	The switch with the index 23 is connected.
TEMP:POV(%N+1)	37	The value of the expression %N+1 is calculated and used as an index.

Table continues on next page

Notation	Value	Explanation
TEMP:PHI	90.0	Alarm is given when the temperature rises above 90.
TEMP:PLI	10.0	or falls below 10.
TEMP:PAL	1	Alarm state prevails.

## 6.3.3 Event handling objects (H)

### 6.3.3.1 Description

Event handling objects are related to the process objects. They define translatable texts that describe the state of the process object and the transitions between the states.

Each process object owns a reference to an event handling object. One event handling object is typically used by several process objects.

### 6.3.3.2 Some event handling attributes

Two event handling object attributes are given below as examples. All attributes are detailed in the Application Objects manual.

ST	State Texts	
	Value:	A text vector containing the text identifiers for the state texts
SX	Translated State Texts	
	Value:	A text vector containing the state texts in the current language

## 6.3.4 Scales (X)

### 6.3.4.1 Description

Scales are related to the process objects, particularly to analog process objects. They define algorithms for the transformation of the digital process values transferred from the stations to the values of the analog units of the corresponding process objects.

Every REAL valued analog process object has a scale name, which defines the scaling algorithm to be used for the transformation. The same scale can be used by several process objects.

### 6.3.4.2 Some scale attributes

Below are two scale object attributes. All attributes are detailed in the Application Objects manual.

LN	Logical Name	
	Value:	text.
SA	Scaling Algorithm	
	Value:	0 = 1:1 scaling 1 = linear scaling 2 = step-wise linear scaling

## 6.3.5 Data objects (D)

### 6.3.5.1 Description

Data objects (datalog objects) register and store calculated or sampled data. They are used for the storage of report data, trend data, data for calculation and control, system configuration data, etc. The data objects can also be used as application variables when there is a need for exchanging data between different objects.

The data registration may be initiated from a program (by means of the commands #EXEC and #EXEC\_AFTER, see [Section 9.2](#)), from a time channel ([Section 6.3.7](#)) or from an event channel ([Section 6.3.8](#)). Each registration is performed according to a SCIL expression and a logging function. Besides the calculated or sampled value, each registration includes a time stamp and a status code.

Every data object can store a chosen number of data registrations (up to 2 000 000). The registrations can be accessed as vectors by means of indices. The oldest registered value has the index number 1.

### 6.3.5.2 Some data object attributes

Some of the most important attributes are listed below. A complete description can be found in the Application Objects manual.

OV	Object Value
	Registered value.
	Value: real or vector of real values.
RT	Registration Time
	Value: time or vector of time values.
OS	Object Status
	Indicates the reliability of the registered data.
	Value: Integer or vector of integers. The status codes listed in the Status Codes manual, for example:
	0 ... OK
	1 ... uncertain
	10 ... no values.

If the object notation lacks an attribute, the attribute is assumed to be OV, except in connection with the command #EXEC or #EXEC\_AFTER ([Section 9.2](#)), where the notation refers to the whole object.

The above mentioned attributes are used indexed. An object notation with these attributes without any index refers to the latest registered value.

### 6.3.5.3 Examples

Some examples of data object notations:

Notation	Value	Explanation
DATA:D	5.6	The latest registered value of the data object.
DATA:DOV	5.6	The latest registered value of the data object.
DATA:DRT		Time of last registration given as time data. Can be converted to calendar time by means of time functions ( <a href="#">Section 10.4</a> ).
DATA:DRT(5..)		The vector formed by the registration times for the datalog object DATA starting with registration number 5.
DATA:DOV(1..(DATA:DLR))		The values of the registered data ranging from the oldest one to the last registration (LR).

## 6.3.6 Command procedures (C)

### 6.3.6.1 Description

The command procedures contain SCIL programs which can be started automatically or by SCIL. They can be used for all kinds of manually or automatically started operations, such as calculations, control operations and reporting. They cannot be used for affecting the user interface. Command procedures can be started in the following ways:

- From SCIL programs with the #EXEC and #EXEC\_AFTER commands ([Section 9.2](#)). (A command procedure can even start itself).
- From time channels ([Section 6.3.7](#)).
- From event channels ([Section 6.3.8](#)).

A command procedure can hold up to 2 000 000 SCIL lines. As a rule, command procedures may not contain user interface related commands (picture commands, Visual SCIL commands or graphical commands, see [Section 9.1](#)). However, a command procedure, or a selected part of it, can be handled as a vector and be executed with the #DO command ([Section 9.2](#)) and the DO function ([Section 10](#)). In these cases, it may contain user interface commands if the #DO command or DO function is executed in a user interface object.

### 6.3.6.2 Some command procedure attributes

Some command procedure attributes are shown below. All attributes are detailed in the Application Objects manual.

TC	Time Channel. The name of the time channel that starts the command procedure. Value: text.
IN	Instruction The program of the command procedure. Value: text vector.
OS	Object Status States how the last execution succeeded. Value: The status codes listed in the Status Codes manual, for example: 0 ... correctly executed 10 ... not executed.

A command procedure notation without an attribute refers to the program, i.e., the IN attribute.

Indices can be used only with the attribute IN where they refer to the line numbers. The IN attribute without indices refers to the whole program.

### 6.3.6.3 Examples

Some examples of command procedure notations:

Notation	Value	Explanation
TASK:C		The program as a text vector.
TASK:COS	0	The program execution succeeded.
TASK:CIN5	"@VAR = 10"	The fifth line of the command procedure TASK.

## 6.3.7 Time channels (T)

### 6.3.7.1 Description

Time channels are used for the automatic start-up of time-bound activities, which can be:

- The registration of data objects.
- The execution of command procedures.

A certain time channel can start several data objects and command procedures. The execution order is determined by the priorities of the data objects and command procedures.

A time channel is activated at the execution time which means that the connected objects are executed. At the initialization time the time channel is initialized, which means that the registration of connected data objects is restarted from the first record. Both initialization and execution can take place at absolute points of time or periodically with a fixed interval. All times are given with an accuracy of one second. Discontinuous time activation is obtained with conditional expressions.

Time channels can be executed also with the #EXEC commands ([Section 9.2](#)) and with event channels ([Section 6.3.8](#)).

### 6.3.7.2 Some time channel attributes

Some time channel attributes are listed below (a complete list is found in the Application Objects manual):

IU	In Use
	States whether the time channel is in use or not.
Value:	0 = not in use 1 = in use

RT	The time of last initialization/activation.
Value:	time vector

The time channels can be used without an attribute only with the #EXEC commands.

Indices (1 or 2) can be used, for example, together with the RT attribute. Index 1 refers to the initialization time and index 2 the activation time.

## 6.3.8 Event channels (A)

### 6.3.8.1 Description

Event channels are used for the automatic start-up of event-bound activities. The events normally originate from the process database, from where the event channels transmit them to objects in the report database which take the consequential actions (calculations, control actions, etc.). An event channel can start the following operations:

- Registration of a data object.
- Execution of a command procedure.
- Activation of a time channel.

Each process object may have only one event channel, but an event channel may be connected to up to 11 command procedures and data objects. At the activation of an event channel, some essential attributes are transmitted as variables from the process object to the connected reporting object. Thanks to this feature, several process objects can share the same event channel.

An event channel is activated by the following events in the activating process objects:

- An alarm comes or goes.
- The warning limits or alarm limits of an analog object are transgressed (provided that the control system, not the RTU, handles the limit value supervision).
- The OV attribute (BI, BO, AI, AO, DI, DO, DB, PC, BS or OE) is changed.
- The OV attribute is updated.

The options are chosen with a process object attribute (AA). Event channels can also be activated with SCIL (the #EXEC commands, [Section 9.2](#)).

In addition, any user-defined attribute can activate the event channel of the object.

### 6.3.8.2 Some event channel attributes

Examples of attributes (a complete list is found in the Application Objects manual):

OT	Object Type
The type of the object to be executed.	
Value:	text.
ON	Object Name
The name of the object to be executed.	
Value:	text.

## 6.3.9 Logging profiles (G)

### 6.3.9.1 Description

Logging profile objects are used to define the connection of SYS600 application database to a Historian database. A Historian database is a high performance time series database designed to store large amounts of real-time data for later analysis and reporting.

The following data from the SYS600 database can be logged to a Historian database:

1. OV (Object Value) attribute of process objects of following types:

- BI (Binary Input)
  - BO (Binary Output)
  - DB (Double Binary)
  - DI (Digital Input)
  - DO (Digital Output)
  - AI (Analog Input)
  - AO (Analog Output)
  - PC (Pulse Counter)
2. OV (Object Value) attribute of data objects of value type real or integer.

In addition to the value itself, also the time stamp and status are logged (attributes RT and OS, respectively). The objects to be logged are specified by linking the object to a logging profile object (attribute GP of process and data objects). The logging profile objects specify where and how the data is logged. For more information about logging profiles, see the Application Objects manual.

## 6.3.10 Event objects (E)

### 6.3.10.1 Description

The event objects are used to start event-bound activities, normally updates, in user interface objects (pictures and Visual SCIL objects).

In pictures, the activation of an event object causes the execution of the statement(s) determined by the #ON command ([Section 9.2](#)) for the specific event object. The #ON commands are valid only for the actual picture, either main picture or sub-picture, where they have been executed. The event causes no action if there is no #ON command in force for the actual event in the picture shown on screen at the event moment.

In dialogs and dialog items, the activation of an event object starts the execution of the event method, if any, defined to be activated by the event object in question.

An event object can be activated in two ways:

- From the process database, so that a change in a process object automatically activates an event object with the same name and index as the process object. The activation takes place independently of what causes the change - a change of state in a station or an assignment in a SCIL program (by the command #SET, [Section 9.2](#)). The coupling of an event object to a process object is optional. The activating attributes are listed in the Application Objects manual, the EE attribute description.
- From a SCIL program (in a picture or a command procedure) by the #EXEC commands ([Section 9.2](#)). In this case the name of the event object can be freely chosen.

If a process object is equipped with an event object, updating in pictures is carried out automatically and immediately when a change occurs. The value of the changed attribute is not transmitted to the picture, nor is any information about which attribute has changed.

The event object notation does not contain any attribute. Event objects have no values, hence they cannot be parts of expressions.

### 6.3.10.2 Example

If the process object

TEMP:P2

is equipped with an event object (EE = 1), the event object

TEMP:E2

is always activated when a change in the process object (for instance the attribute AI) occurs in the process database.

## 6.3.11 Variable objects (V)

### 6.3.11.1 Description

The variable objects serve as temporary storage places for attributes. They are used to compose lists, for example, alarm and event lists, to browse through the object properties, to copy objects, to create and modify objects, etc.

A variable object is at the same time both an object and a global variable of list type ([Section 5.11](#)). The list as a whole is handled as a variable with the same name as the object ([Section 7](#)). The attributes in the list are accessed with a variable object notation.

A variable object can be created and assigned attribute values in two ways:

- By creating the variable object with the #CREATE command ([Section 9.2](#)) and assigning it attribute values with a list function or with the #SET command ([Section 9.2](#)). In this case, the attribute names may be arbitrary, up to 63 characters.
- By assigning a global variable a value of list type ([Section 7.2](#)). A variable object with the same name as the variable is formed. The variable object gets all the attributes of the list expression.

A variable object notation must always contain an attribute. Unlike other object types, the names of variable object attributes can contain up to 63 characters. If the attribute is of vector type, the vector elements are identified by indices. An object notation without an index denotes the whole vector. If the attribute name contains more or less characters than two, the index must be enclosed by parentheses.

As global variables ([Section 7](#)), variable objects are accessible within the SCIL context they are created in.

### 6.3.11.2 Examples

The statement

```
@VAR = PROD_QUERY(20)
```

assigns the variable VAR the list value returned by the function PROD\_QUERY ([Section 10](#)). The variable may be accessed as variable object VAR:V.

Some variable object notations:

Notation	Value	Explanation
VAR:VLN1	"SWITCH"	The first object name in the list.
VAR:VOV1	0	The first object value in the list.
VAR:VLN(1 .. 10)		The first ten object names in the list.



Variable objects are more or less obsolete. Attributes of list type variables may be accessed in a more powerful way by using data component access described in [Section 5.11](#). For example, VAR:VLN1 may be written as %VAR.LN(1), when read, or @VAR.LN(1), when written.

## 6.4 User interface objects

### 6.4.1 Visual SCIL objects

#### 6.4.1.1 General

The Visual SCIL objects correspond to the dialogs, dialog items and images designed in the Dialog Editor or created with SCIL. There are approximately 40 types of Visual SCIL objects which could be grouped as follows:

- Dialogs. There are two types:
  - Ordinary dialogs
  - Main dialogs
- Compound dialog items. These are dialog items which may contain other dialog items:
  - Containers: visible or invisible boxes containing other dialog items.
  - Picture containers: containers that can contain pictures.
  - Menu bars, menus and menu items: menu bars can contain one or more menus, menus can contain menus and menu items.
  - Notebooks, notebook pages and notebook items.
- Simple dialog items, such as buttons, toggle buttons, texts, lists, etc. These dialog items cannot contain other objects.
- Images.

In SCIL, the visual SCIL object types are referred to by names starting with VS\_. For instance, buttons are named VS\_BUTTON. The Visual SCIL Objects manual lists and describes all the Visual SCIL object types.

Generally, the dialogs, dialog items and images are designed in the Dialog editor, though they can also be created directly with SCIL. Dialog and dialog items created in the dialog editor are stored in files from where the SCIL command .LOAD loads them in the memory as a visual SCIL object with a specific name. Objects created with .CREATE are not stored in files.

The SCIL commands for handling Visual SCIL objects are detailed in [Section 9.2](#).

Unlike system and application objects, the Visual SCIL objects are not global. They are known and accessed only within the dialog system where they are created or loaded.

#### 6.4.1.2 Dialog systems

The Visual SCIL objects are arranged in hierarchical dialog systems with a main dialog or picture container at the top. Each main dialog and each dialog system creates a new dialog system. The object hierarchy is important when referencing the objects.

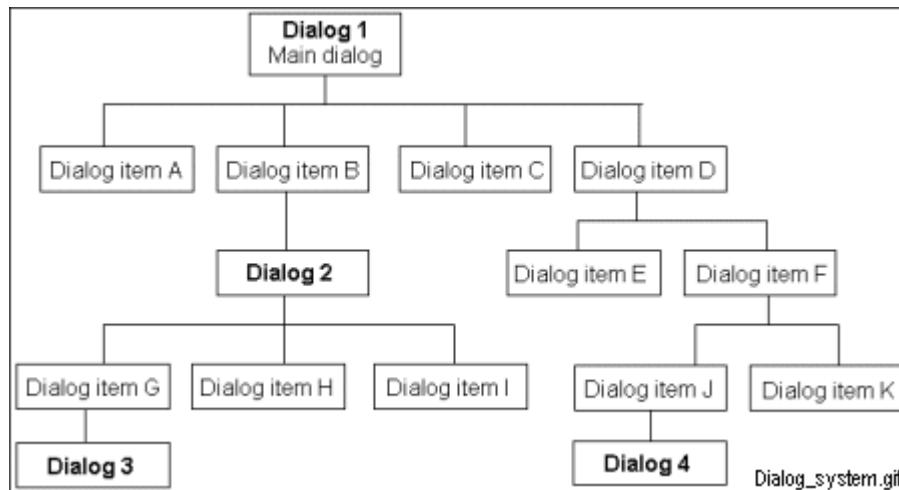
Objects containing another object are called parent objects, and the contained objects are called child objects. Child objects are dependent upon their parent objects. When the parent objects are loaded, the child objects are loaded as well, and when the parent objects are deleted (with the .DELETE command), the child objects are also deleted.

When loading a dialog or a dialog item with the .LOAD command, it becomes a child object of the loading object (unless object path is given, see below). When loading a main dialog or picture container, it creates a new dialog system.

[Figure 8](#) shows an example of a Visual SCIL dialog system with four dialogs. Dialog 1, which is the main dialog, contains four dialog items. Dialog item B loads dialog 2. This dialog contains three dialog items, one of which (object G) loads dialog 3. Dialog item D contains two dialog

items, and dialog item F contains two items. For instance, D could be a menu bar containing menus, which contain menu items (menu options). Dialog item J loads dialog 4.

The dialog items A .. D are child objects of the main dialog. Dialog 2 is the child object of item B and the parent object of the dialog items G .. I. Dialog 4 is the child object of item J which is the child object of item F, and so on.



*Figure 8: An example of a dialog system.*

#### 6.4.1.3 Attributes and methods

The attributes of the Visual SCIL objects specify their properties and functions, for example, visibility, position, text contents, behavior, color, fonts, etc. Some attribute values can be set in the dialog editor, some only with SCIL, and some both in the editor and with SCIL.

The dynamic changes of the dialogs and dialog items are obtained by changing their attribute values with the SCIL commands .SET and .MODIFY.

Each object has a number of predefined attributes with predefined names, data types and meanings. The predefined attributes of each object type are listed and described in the Visual SCIL Objects manual. In addition, the objects can be given an arbitrary number of user defined attributes with the .LOAD, .CREATE and .MODIFY commands.

Each dialog and dialog item may have a number of methods which are programs for various purposes, for example, cyclically executed programs, programs executed on certain events (event objects), programs started by a user operation, and programs started by SCIL.

The programs started by SCIL may be predefined or user defined. They are started by a method call of the format described below.

#### 6.4.1.4 Visual SCIL object references

When designing a dialog or dialog item in the Dialog Editor, each item may be given a name. This name will be the Visual SCIL object name of the item when it is included in a dialog or dialog item loaded with the .LOAD command. The loaded dialog or dialog item is given a Visual SCIL object name with the .LOAD command. Objects created with .CREATE are given object names with this command.

Object references are used when loading, creating and deleting objects, and in the attribute references and method calls. When referring to an object one level below, the object name is sufficient. When referring to an object that is more than one level below, object path should be given as follows:

name\name\name\.... \name

where each 'name' is a Visual SCIL object name.

The following predefined path names can be used:

ROOT	The main dialog or picture container on top of the dialog system.
PARENT	The parent of the object.
THIS	The object in question.

PARENT and ROOT are the only possibilities for accessing objects on a higher level in the hierarchy.

#### 6.4.1.5 Attribute references

Attributes are referenced with the following notations:

[object].attribute[component]\*

where 'object' is an object reference, 'attribute' is the name of the attribute and 'component' addresses a component of structured data, see [Section 5](#). If the attribute or method is referenced from a method within the same object, no object reference is needed.

Attribute references can be used as operands in expressions. They are also used together with the .SET command to achieve a change of the attribute. The attribute references can be expanded like variables, see [Section 7](#).

Example:

.SET MY\_BUTTON.\_TITLE = "OK"

This statement sets the label of the button (the visual SCIL object MY\_BUTTON) to OK, which is immediately shown on screen if the button is shown.

#### 6.4.1.6 Method calls

Method calls start the execution of methods, both predefined methods and user defined methods. Method calls have the following format:

[object].method [(argument\_list)]

where

'object'	is an object reference. Not needed when referencing methods in the same object or one level below.
'method'	is the method name.
'argument_list'	Arguments, which may be any SCIL expressions given as a list of SCIL expressions separated by comma and enclosed in parentheses. Up to 32 arguments may be given.

By using the SCIL command #RETURN, a method can be programmed to return a value to the calling program. A method call that returns a value can be used as an operand in an expression.

## 6.4.2 Pictures

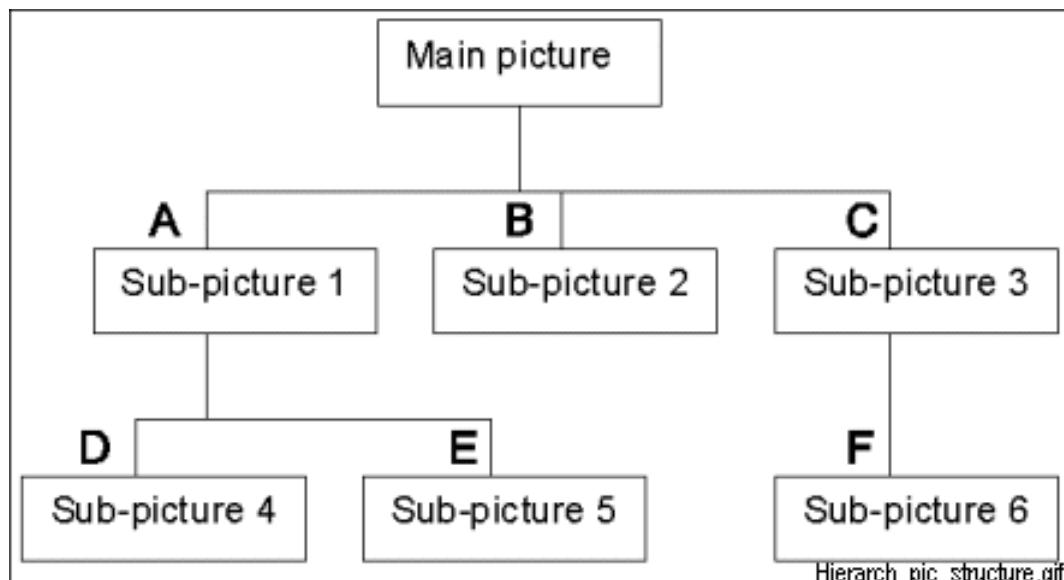
### 6.4.2.1 General

The pictures and picture components were briefly described in [Section 4](#). The Picture Editing manual describes how to build and program the pictures. This section discusses how to reference pictures and picture elements from SCIL.

In SCIL, the pictures and windows are handled by the picture handling commands described in [Section 9.4](#). In addition, the windows may have attributes which are accessed by a window attribute reference. The named programs of the pictures are executed by a named program call.

### 6.4.2.2 Picture hierarchy

A picture may contain a number of sub-pictures in the shape of pictures shown in windows and picture functions. These sub-pictures in turn may contain sub-pictures. A picture and its sub-picture form a picture hierarchy with the main picture (total picture) on the top. [Figure 9](#) shows an example of such a hierarchy. A, B, C, D, E and F denotes windows and picture functions.



*Figure 9: A hierarchical picture structure. The subpictures are picture functions or pictures shown in windows. The letters A .. F denotes the names of windows and picture functions.*

A picture containing sub-pictures is the parent of its sub-pictures and the contained sub-pictures are children. This parent-child relationship is important in determining how the windows are shown in relation to other windows. It is also important when referencing windows and named programs.

### 6.4.2.3 Picture Paths

When referencing windows and named programs, a picture path may be used. A picture path states in which picture, main picture or sub-picture, the window or named program should be searched. A picture path has the following format:

name/name/name/ ... /name

where each 'name' is the name of the window or picture function containing the next picture in the hierarchy counting from the picture where the command is issued. (Back slash (\) could be used instead of forward slash (/).) The main picture can be referenced with the predefined name ROOT. If the picture is included in a dialog system (see [Section 6.4.1](#)) ROOT also refers to the picture container where the picture is included.

If no picture path is given, the window or named program is first searched from the same picture as where the reference is issued. If not found there, it is searched from the parent picture and so on up to the main picture. Hence, when referencing windows and programs within the same picture or its parent picture, no path is needed.

For example, the picture path for referencing a named program SAMPLE in sub-picture 4 in [Figure 9](#) from the main picture would be:

A/D

If not found in sub-picture 4, the program is searched in sub-picture 1 and then in the main picture.

#### 6.4.2.4 Picture object attributes

The programmer can assign the windows and picture functions attributes for various purposes using the .SET command, see [Section 9.2](#).

The window and picture function attributes are referenced according to the following format:

[picture].attribute[component]\*

where 'picture' is the picture path, 'attribute' is the name of the attribute and 'component' addresses a component of structured data, see [Section 5](#).

An attribute reference can be used as an operand in SCIL expressions. The operations allowed depend on the data type of the attribute (see [Section 8](#)).

The picture object attributes can also be expanded in the same way as variables (see [Section 7](#)).

Examples:

```
.SET PIC_FUNC_1/WINDOW_2.RESIZED = TRUE
.SET .COUNTER = 0
.SET .COUNTER = .COUNTER + 1
```

#### 6.4.2.5 Named program calls

Each picture can have a number of named programs which are started with the following program call:

[picture].program[(argument\_list)]

where

'picture'

is the picture path to the object containing the named program, see above.  
If omitted, 'program' is searched for in the picture containing the program call. If not found there, it is searched in its parent picture, and so on.

'program'

The name of the named program.

'argument\_list'

Arguments given as a list of SCIL expressions separated by commas and enclosed in parentheses. Up to 32 arguments may be given.

When encountered in a program, the program call starts the execution of the named program in the given picture. The subsequent statement in the calling program is not executed until the named program has been executed to the end. If no program with the given name is found, an error status is produced. Named programs can be executed from any program in the dialog system. They are also used as call back programs of Motif widgets.

By using the SCIL command #RETURN (see [Section 9.2](#)) in the named program, it can be programmed to return a value to the calling program. A named program call that returns a value can be used as operand in expressions.

**Example:**

```
A/B.CREATE_ERROR_DIALOG
Executes the named program CREATE_ERROR_DIALOG of window B which is a child window of window A.
```

```
@A = .MY_NAMED_PROGRAM (125,%V+6,"ABC")
The program MY_NAMED_PROGRAM is executed and its return value is assigned to the variable A.
```

## 6.4.3 Predefined VS object, window and picture function methods

Every Visual SCIL object, window and picture function has the following predefined methods that are used to postpone a SCIL program execution.

### 6.4.3.1 \_FLAG\_FOR\_EXECUTION(name, program [, delay])

Queues a SCIL program to be executed.

'name'	Text value used as the identifier of the flagged execution. An empty name is allowed.
'program'	Text or text vector, the program to be executed.
'delay'	Real value indicating the minimum amount of seconds to elapse before the program is executed. Default value is 0.
Return value:	No return value.

This method is identical to \_QUEUE\_FOR\_EXECUTION (see below) with the following exception: If \_FLAG\_FOR\_EXECUTION method of an object is called twice using the same 'name', the first flagged execution is cancelled (if still queued).

### 6.4.3.2 \_QUEUE\_FOR\_EXECUTION(program [, delay])

Queues a SCIL program to be executed.

'program'	Text or text vector, the program to be executed.
'delay'	Real value indicating the minimum amount of seconds to elapse before the program is executed. Default value is 0.
Return value:	No return value.

This method is used when the programmer wants to execute a SCIL program at a later point in time when the process would otherwise be idle. The program to be executed is given as parameter in the method call. There is no way to exactly know when in time the program is executed. A minimum time that has to expire can however be defined in seconds as the second parameter. The maximum number of queued programs per object is 100. If this limit is exceeded all queued executions are removed and a SCIL error is raised.

Example:

The example shows how to ensure that the blocking cursor of a dialog is switched off after a long program execution even if an error occurs that interrupts the normal program flow.

```
.set my_dialog._busy = true
my_dialog._queue_for_execution(".set this._busy = false")
;long SCIL processing here
...
```

## 6.4.4 Predefined VS object, window and picture function attributes

Every Visual SCIL object, window and picture function has the following predefined attributes that provide information of various properties of the object. All these attributes are read-only.

### 6.4.4.1 \_ATTRIBUTE\_NAMES

The attributes listed in alphabetical order.

Value type:	List.
Value:	Two attributes:
	USER_DEFINED Text vector containing the names of user-defined attributes.
	PREDEFINED Text vector containing the names of the attributes defined by the object class.

### 6.4.4.2 \_CHILD\_OBJECTS

The names of the child objects.

Value type:	Text vector
Value:	The names of the immediate children of the object listed in the order of creation.

### 6.4.4.3 \_COMPILED

Defines whether the SCIL programs of the object are compiled or not.

Value type:	Boolean	
Value:	FALSE	Not compiled (default)
	TRUE	Compiled

When the attribute is set to TRUE, the object and all its child objects that are loaded together with it, are recursively flagged to be compiled. The methods of the objects are not compiled at once, but instead they are compiled when called for the first time. If the compilation fails, the uncompiled source code is silently used instead.

When the attribute is set to FALSE, the \_COMPILED flag of the object and its child objects are recursively cleared, and the compiled methods (if any) are decompiled, i.e. their compilation result is discarded.

The \_COMPILED attribute may be set any time (even from outside the object), but it is good practice to insert the SCIL statement

```
.set THIS._compiled = TRUE
```

in the CREATE method of the object. This is to state that the object has benefit from the compilation and is comprehensively tested to run as compiled.

#### **6.4.4.4    \_FILE\_REVISION**

File revision.

Value type:	Text
Value:	The FILE_REVISION attribute of the picture file (.pic) or the Visual SCIL object file (.vso) the object was loaded from.

The value is an empty string if the object has been created on-the-fly.

#### **6.4.4.5    \_OBJECT\_CLASS**

The class of the object.

Value type:	Text
Value:	The name of the class of the object

If the object is a window or picture function, the attribute has value "WINDOW" or "PICTURE\_FUNCTION", respectively.

#### **6.4.4.6    \_OBJECT\_NAME**

The name of the object.

Value type:	Text
Value:	The name of the object

#### **6.4.4.7    \_OBJECT\_PATH**

The path of the object.

Value type:	Text
Value:	The complete object path starting from the root, For example, "ROOT\NBK_BOOK\NBP_PAGE\TRE_TREE"

#### **6.4.4.8    \_SG\_GEOMETRY**

This attribute is applicable only for windows and picture functions.

Value type:	List.
Value:	Six attributes:
	X                    X position of window/pf relative to part picture / picture function.
	Y                    Y position of window/pf relative to part picture / picture function.
	ABSOLUTE_X        X position of window/pf relative to ROOT.
	ABSOLUTE_Y        Y position of window/pf relative to ROOT.

Table continues on next page

W	Width of window/picture function in semigraphical characters.
H	Height of window/picture function in semigraphical characters
W and H are 0 if window is not shown.	

#### 6.4.4.9 SOURCE\_FILE\_NAME

The name of the file where the object is loaded from.

Value type:	Text.
Value:	The full operating system format name of the vso-file where the object has been loaded from or the full name of the pic-file where the window or picture function has been read from.

The value is an empty string, if the object is created on-the-fly, i.e. using .CREATE, !WIN\_NAME or !WIN\_CREATE command.

#### 6.4.4.10 VARIABLE\_NAMES

The names of the variables seen by the object.

Value type:	Text vector.
Value:	The names of global SCIL variables defined in the SCIL context that the object belongs to.

The names are listed in alphabetical order.

## 6.5 Files

This section describes the principles of naming files in SCIL language, and the different file types accessible by SCIL programs.

Here, file type refers to the interpretation of its contents. The three file types supported by SCIL programming environment are the following:

- Text files
- Binary files
- Keyed files

The file types are further described in the following sections. File management functions, that handle entire files with no interpretation of the contents, are described in [Section 10.17](#).

### 6.5.1 File naming

The files and directories used in SCIL language may be identified in several different ways, both operating system dependent and independent ways. The SCIL functions and commands that take a file a directory name argument, accept all these identifications unless otherwise noted in their description.

In the following, the term SYS600 root is referred to. It is the root directory of the current SYS600 installation. In Windows, it is normally the directory C:\SC\, but the drive may be changed during installation.

File and directory tags are abstract, operating system independent identifiers, that are formed from textual file and directory names by means of file management functions described in [Section 10.17](#).

A directory may be identified in one of the following ways:

- "/dir1/dir2/.../name/" names a directory relative to SYS600 root in an operating system independent way, for example, "/APL/TIPPERARY/APL\_/".
- By an operating system file path, for example, "D:\SC\APL\TIPPERARY\APL\\_".
- By an operating system independent directory tag created from a SCIL directory name or an operating system directory name.

A file may be identified in one of the following ways:

- "path/name" identifies a file contained in a logical path in an operating system independent way. The logical paths are simple names given to a directory or a search list of directories. See [Section 9.2.4](#) for more information on logical paths.
- "name" identifies a file contained in the default directory of the SCIL context or in the logical path derived from the file name. The default directory is normally the directory PICT below the root directory of the SCIL application, but may be set by SCIL in the VS\_MAIN\_DIALOG object. Outside the VS main dialogs, the so-called 4-letter rule is further applied: A file is first searched for in the logical path named according to the 4 first letters of the file name. For example, file APL\_PROCES.PRD is searched for using the logical path "APL\_".
- "/dir1/dir2/.../dir/name" names a file relative to SYS600 root in an operating system independent way, for example, "/APL/TIPPERARY/APL\_/FILE.EXT"
- By an operating system file path, for example, "D:\SC\APL\TIPPERARY\APL\_\FILE.EXT".
- By an operating system independent file tag created from a file name given by any of the textual ways above.

The file or directory name ('name' above) may be of any length and contain any characters allowed by the operating system. However, as far as portability is concerned, names consisting of letters, digits, underscores and a dot (as a separator of the file name extension) only are recommended.

## 6.5.2 Text files

Text files are sequential files organized as lines of text data. When a text file is read, the text is supposed to be encoded via the Active Code Page (ACP) of the operating system unless the actual text is preceded by a Byte Order Mark (BOM). UTF-8 and UTF-16 (both little and big endian) BOM's are supported. By default, text files are written in UTF-8 encoding with a BOM. Optionally, the functions WRITE\_TEXT and WRITE\_COLUMNS are capable to write ACP encoded text as well.

The lines are terminated by CR/LF (Carriage Return / Line Feed) character pair. When a text file is read by SCIL, a single LF character is accepted as a line terminator and the terminator of the last line of the file is not required.

Text file is read by the TEXT\_READ or READ\_COLUMNS function and written by the WRITE\_TEXT or WRITE\_COLUMNS function. The entire file or a bulk of lines is read and written as a text vector by these functions. Line lengths up to 65 535 characters are supported by the functions.

An obsolete function READ\_TEXT is capable of reading only 255 character lines. It is supported for upward compatibility, but new applications are encouraged to use the TEXT\_READ function instead.

Parameter files are special purpose files of predefined format similar to INI files of used by various utilities of Windows operating system. They are designed to save settings of tool programs from invocation to another. SCIL functions READ\_PARAMETER, WRITE\_PARAMETER

and DELETE\_PARAMETER implement the handling of parameter files. The format of parameter files is described in Appendix B.

While a text file is being written, no other SCIL program may read or write the file. Consequently, if a text file is used to pass data from a SCIL program to another SCIL program that may be executing in parallel, the access of file should be synchronized between the programs, for example using function FILE\_LOCK\_MANAGER.

For descriptions of functions mentioned above, see [Section 10.16](#).

## 6.5.3 Binary files

Binary files are byte sequences of any length with no interpretation of structure. They can be used to exchange data between a SYS600 application and any foreign application. Within SYS600 they are used to store compiled SCIL programs and download/upload data of process objects of File Transfer type.

A binary file is read by READ\_BYTES function and written by WRITE\_BYTES function. The entire file or up to 8 388 600 bytes of data is read and written as a byte string by these functions.

While a binary file is being written, no other SCIL program may read or write the file. Consequently, if a binary file is used to pass data from a SCIL program to another SCIL program that may be executing in parallel, the access of file should be synchronized between the programs, for example using function FILE\_LOCK\_MANAGER.

For descriptions of functions mentioned above, see [Section 10.16](#).

## 6.5.4 Keyed files

### 6.5.4.1 General

Keyed files are random access files that consist of data records. The maximum length of a data record is 508 characters.

The beginning of the data record contains the key of the record. The key uniquely identifies the record within the file. The length of the key is defined when a keyed file is created. The maximum length of the key is 253 characters. The data records are logically ordered by the keys. The first character of the key is the most significant in the ordering, the last character is least significant. Hence, if the key contains a name, the records are alphabetically ordered.

Each data record must contain the key. Consequently, the minimum length of the data record is the length of the key.

A data record can be read from a file by specifying its key, or the file may be browsed forward or backward in the order specified by the record keys.

When a data record is written into the file, it is always positioned according to its key. If the same key already exists in the file, the old data record is overwritten by the new one. The length of record may change when it is rewritten.

Data records may also be deleted one by one by specifying the key to be deleted.

Concurrent access of keyed files by several SCIL programs is internally synchronized. Therefore, they suit well for exchanging real-time data between SCIL programs. More than one program may safely write and read a keyed file simultaneously.

### 6.5.4.2 Implementation

The keyed files are implemented as a number of data blocks, index blocks and a status block.

The **data blocks** contain the data records of the file. Each data block contains a number of consecutive (in the order specified by the key) data records organized in the order of the key.

The **index blocks** make up the index of data blocks in order to achieve fast access of data. An index block contains a set of (key, block number) pairs, where the key is the first key of the data block addressed by the block number. The pairs contained in an index block are again organized in the order of the key. When the file grows, more than one index block is needed. Upper level index blocks are then created. In the upper level index blocks, the block number addresses a lower level index block instead of a data block.

The **status block**, which is the first block of the file, contains book-keeping information of the file, such as the length of the key and address of the topmost index block.

When a new data record is written into the file (or an existing one is expanded), the index is first used to locate the would-be data block for the record. If there is enough free space in the block, the data is written into the block. Otherwise, the neighboring data blocks are examined. If there is enough free space in the neighboring blocks to rearrange the old records and the new one into these three data blocks, it is done. Otherwise, a new data block is created and its key is inserted in an index block. The same procedure is then repeated for the index block.

When a data record is deleted from a file (or an existing one is shortened), the free space of a data block grows. When it is possible to combine the data block with its neighbors, it is done and the freed block is inserted in the free block list of the file for later re-use.

There are two internal implementations of keyed files:

- Version 1 file format uses 512-byte blocks. The size of the file is limited to 32 megabytes.
- Version 2 file format, first introduced in MicroSCADA revision 8.4.4, uses 4 kilobyte blocks and has no file size restrictions.

Function KEYED\_FILE\_MANAGER (see [Section 10.16](#)) may be used to convert version 1 files to version 2 and vice versa.

The same function may be used to two other maintenance purposes of keyed files:

- Subfunction COMPACT makes a reorganized copy of a keyed file. The new file is written in its most compact form and the free space in the file is minimized. The resulting file may be smaller and faster to access.
- Subfunction REBUILD makes a copy of a keyed file where the index structure is rebuilt from the scratch. The source file is scanned sequentially, the data records found in the data blocks are written to the output file and a new index is created for them. The index blocks of the source file are skipped. This subfunction should be used, when the structure of a keyed file gets corrupted for a reason or another, for example because of power fail of the computer. A corrupted file may often be recognized by spurious FILE\_INCONSISTENT (5015) errors got when reading the file.

### 6.5.4.3 Use

Keyed files are widely used within SYS600 to implement various system files, including:

- Process database file APL\_PROCES.PRD
- History database files APL\_yymmdd.PHD and APL\_yymmdd.PHI
- Report database files APL\_REPORT.nnn
- Picture files \*.PIC
- Representation library files \*.PIR
- SCIL database files \*.SDB

For application specific purposes, keyed files may be used by file handling commands described in [Section 9.2.5](#).

## 6.6 SCIL databases

### 6.6.1 General

A SCIL database is a data file for general purpose. It is internally structured to store any data structure that may be created by SCIL language. The file is divided into sections. Each section has a value that may be of any SCIL data type. The size of a SCIL database file is limited only by the available disk space.

The structure of SCIL database is optimized for fast access. The component to be read is located quickly by using indices without having to search through the file. Similarly, writing a new value to a component rewrites only a minimal part of the file.

The concurrent use of SCIL database by two or more SCIL programs is internally synchronized. In addition, each read and write operation is atomic. For example, if one SCIL program writes a list type data structure into the database, the concurrent readers see all the new attribute values or none of them. Because of the strict access discipline, SCIL databases suit perfectly for sharing SCIL data in real-time between various SCIL programs of the application (in pictures, Visual SCIL objects, command procedure objects etc.).

SCIL databases and parameter files (so called ini files handled by functions READ\_PARAMETER, WRITE\_PARAMETER and DELETE\_PARAMETER) have some similarities, but at the same time some important differences. For some comparison of SCIL Database and parameter files, see the table below:

Property	SCIL database	Parameter file
Structure	Indexed	Text lines
Data encoding	Binary, cannot be viewed with text editor	Plain text, can be viewed with text editor
Maximum size	No limit	Limited by the SCIL vector length
Data item data types	All SCIL data types	Text only
Size of data items	Unlimited	About 65 000 characters
Speed of access	Fast	Slow, when the file is big
Concurrent access by several SCIL programs	Yes	No

### 6.6.2 Versions

There are two slightly different implementations of SCIL database files:

- Version 2 implementation was introduced in MicroSCADA 8.4.5.
- Version 3 implementation was introduced in MicroSCADA 9.0. For the user's point of view, it is equivalent to version 2, but internally it has been optimised for faster access.
- (Version 1 implementation was never used in any released product.)

The version of a SCIL database file is seen as an attribute of the result of the DATA\_MANAGER("OPEN") function call, see [Section 10.16](#).

When MicroSCADA 9.0 or later is used, version 3 files are created by default. If a 9.0 system or later is frequently used to create SCIL databases for an 8.4.5 system, this behavior may be overridden by the revision compatibility switch CREATE\_VERSION\_2\_SCIL\_DATABASES in the

application attribute APL:BRC, see the System Objects manual, or in the SCIL function REVISION\_COMPATIBILITY, see [Section 10.9](#).

When running MicroSCADA 9.0 or later, SCIL database files may be converted from version 2 to 3 (for faster access) or vice versa (for 8.4.5 compatibility) by using the DATA\_MANAGER("COPY") function, see [Section 10.16](#).

### 6.6.3 Access

The SCIL database is divided into sections (the same way as the parameter files). The names of the sections are free texts, they may contain any number of characters and are case-sensitive. A zero-length section name is allowed.

A section has a value, which may be of any SCIL data type. The value may be read and written as a whole or only a component of it is accessed at a time.

In most cases, there is a need for more structure than the sections provide. An obvious way of getting more structure is to define the section as a list value, whose attributes may be of any data type.

If the database is used by several SCIL programs at the same time, the structure of the data should be carefully designed. See an example of this below.

If a section contains a value of a process object along with its time stamp and status, one possibility is to define three attributes in the section, for example OV, RT and OS, respectively. When a SCIL program updates the value, it has to do 3 writes into the database (unless it rewrites the whole section).

Now the reader of the data may encounter a problem. On one hand it is possible, for example, that it reads the new value of OV but the old values of RT and OS (if it runs faster than the writer). On the other hand, it might be that it reads the old value of OV but the new values of RT and OS (if it runs slower).

A solution to this problem is to define only one attribute, for example VALUE, and make it a list of attributes OV, RT and OS. This attribute may now be read and written in one atomic operation.

SCIL databases are managed and accessed by the SCIL function DATA\_MANAGER, which is described in [Section 10.16](#).

### 6.6.4 Use

SCIL databases are suitable for storing application specific data and sharing it in real time between concurrent processes. The following four SCIL databases are used (and automatically created) by the base system:

- Common purpose application database APL\_DATA.SDB
- Application text database APL\_TEXT.SDB
- System text database SYS\_TEXT.SDB
- OPC Name Database APL\_OPCNAM.SDB

For fastest possible access, these files are kept permanently open by the base system.



# Section 7 Variables

This section describes the use of variables in SCIL:

<a href="#">Section 7.1</a>	<a href="#">General: Variable names and scope of variables</a>
<a href="#">Section 7.2</a>	<a href="#">Local variables</a>
<a href="#">Section 7.3</a>	<a href="#">Global variables</a>
<a href="#">Section 7.4</a>	<a href="#">Using variables</a>
<a href="#">Section 7.5</a>	<a href="#">Predefined picture variables</a>

## 7.1 General

Contrary to constants, the variables have no fixed values. A variable is a name which may be assigned any value. After the assignment, the variable name represents this value within the scope of the variable.

Any SCIL data type ([Section 5](#)) is allowed as a value of a variable. When speaking of the data type of a variable, the data type of its current value is meant. The value of a variable, also its data type, may be changed at any time by new assignment.

### 7.1.1 Variable names

The variable names may be freely chosen in accordance with the rules in [Section 4.2](#). The variable names may contain up to 63 characters.

### 7.1.2 Scope of variables

There are two types of variables in SCIL language: local and global variables.

The local variables are temporary variables that are available only within the SCIL program where they are declared. They exist only while the SCIL program is executing.

The global variables are permanent variables that are available for any program executing within the SCIL context where they were created. They exist during the entire life time of the SCIL context (see below).

## 7.2 Local variables

Local variables are declared by #LOCAL statement (see [Section 9.2](#)) at the beginning of a SCIL program and destroyed automatically when the program terminates.

Local variables are visible only in the SCIL program that contains the declaration. The same variable name may be declared by several SCIL programs, each declaration denotes a new variable. If a SCIL program recursively calls itself, each invocation has its own local variables.

Within the SCIL program, a local variable is referenced simply by its name, no special characters are needed.

Arguments of a SCIL program act as read-only local variables in the program, see #ARGUMENT statement in [Section 9.2](#).

## 7.3 Global variables

Unlike local variables, global variables need no declaration. A global variable is created automatically when it is assigned a value first time. It is deleted when the SCIL context containing the variable is deleted, see below. Global variables may also be deleted explicitly with the #DELETE command.

A global variable is visible for all SCIL programs executing in the SCIL context during its entire life time.

In a SCIL program, a global variable is referenced by its name prefixed by character %, when reading its value, or by character @, when assigning it a new value.

### 7.3.1 SCIL contexts

A SCIL context is actually an internal data structure that contains the global variables, logical paths and logical representation libraries defined by SCIL programs executed within the context. A SCIL context is created when certain objects start their execution and is deleted when they terminate.

The following objects execute in their own SCIL context:

- A picture (whether printed or displayed on the screen). All programs, windows and sub-pictures within the same picture share the SCIL context of the picture. A variable defined in a window picture, for example, can be used in the main picture, and vice versa. If window specific variables are desired, use window attributes instead of variables (see [Section 6](#)).
- A dialog system. All dialogs and pictures within the same dialog share the SCIL context of the main dialog or the picture container. Hence, for example, variables defined in the main dialog can be used in all its child dialogs and dialog items. If either dialog or dialog item specific variables are desired, use the user defined attributes (see [Section 6](#)).
- A command procedure and data object started with #EXEC or #EXEC\_AFTER or by an event channel.
- A time channel. Data objects and command procedures run by the time channel share the SCIL context of the time channel. Hence, variables defined in command procedures may be used by command procedures and data objects executed later by the same time channel.

A SCIL context is normally created as empty: no variables, logical paths nor logical representation libraries defined. There are the following exceptions to this rule:

- Objects activated by #EXEC, #EXEC\_AFTER, #PRINT or #LIST command inherit the logical paths and logical representation libraries of the activating SCIL context.
- Argument variables may be passed to objects by means of #EXEC, #EXEC\_AFTER and #PRINT commands (see [Section 9.2](#)).
- Event channels and format pictures activated by process events and format pictures printed by #LIST command start with a set of variables created from attribute values of the process objects, so called ‘snapshot variables’.
- Pictures have a few predefined variables created automatically (see [Section 7.5](#)).

## 7.4 Using variables

### 7.4.1 Variable assignment

Variables are assigned values with an assignment statement of the following syntax:

**[@]V[component]\* = expression**

where 'V' is the variable name. If it is prefixed by @, a global variable is accessed, otherwise a local variable.

Each 'component' addresses a component of structured data (an element or a range of elements of a vector or an attribute of a list). See [Section 5.11](#) for component access.

The expression, which may be of any data type, is evaluated and the value is assigned to the variable or to its component.

(If the expression is a text constant, a global variable assignment may also be written without the equal sign as follows:

**@V character string**

which is the same as the statement @V = "character string", except that in the first case lower case letters are converted to upper case. This is an obsolete feature that is no longer recommended.)

### 7.4.2 Examples

Statement	Explanation
@VAR = ABC:PAI	The value of the process object is read from the process database and assigned to the variable.
TEXT = "ABC:PAI"	The object value is not read. The variable gets the text value "ABC:PAI".
@LIST = PROD_QUERY(20)	The variable LIST is assigned the list value formed by function PROD_QUERY.
A.EXISTS = FALSE	Attribute EXISTS of list variable A is set to FALSE.
@V = DATA:D(1 .. 20)	Variable V becomes a vector containing the first 20 registered values of the data object DATA.
V(1 .. 5) = D1:D(1 .. 5) + D2:D(1 .. 5)	The first five elements in variable V are assigned the values of the sums of the first five registered values of the data objects D1 and D2.
A = (5, 4, OBJ:POV3, CLOCK)	The variable A becomes a vector of four elements.

### 7.4.3 Using variables in expressions

After a variable has been assigned a value, this value is referred to as:

**[%]name[component]\***

where 'name' is the variable name. If it is prefixed by %, a global variable is accessed, otherwise a local variable or an argument.

Each 'component' addresses a component of structured data (an element or a range of elements of a vector, or an attribute of a list). See [Section 5.11](#) for component access.

This notation can be used as an operand in expressions. The data type of the variable determines which operations may be carried out on it ([Section 8](#)).

## 7.4.4 Examples

Statement	Explanation
@NEW = 30 * %OLD	The variable NEW is assigned the value of the variable OLD multiplied by 30.
A.COUNT = A.COUNT + 1	The COUNT attribute of list variable A is incremented.
!SHOW WIN VAR	The whole vector is shown in the window WIN, which must be of the type MULTIFIELD, BAR or CURVE.
@S = %A(1) + %A(2)	The sum of the elements one and two in variable A.

## 7.4.5 Variable expansion

Variables can also be used for forming text strings and names. By including a variable in a name or text string, different contents can be assigned to the text or name depending on the context. To use a variable as a part of a text or name, enclose the variable name by quotes as follows:

'name[.attribute]'

The value of the variable 'name' or the value of its attribute 'attribute' is regarded as a text constant that replaces the quote notation. The value must be of data type integer, real, text, boolean (0 for FALSE, 1 for TRUE is expanded) or time (yy-mm-dd hh:mm:ss is expanded). If a local variable by name 'name' exists, it is used for expansion, otherwise global variable by that name.

This way of using variables is called variable expansion. Not only variables but also window attributes and Visual SCIL object attributes can be expanded, provided that they are of proper data type. In this case 'name' is a window or a Visual SCIL object reference.

## 7.4.6 Examples

Statement	Meaning
LN = "P_METER" IX = 22	The variable LN is assigned the text value "P_METER" and the variable IX value 22
PRESSURE = 'LN':PAI'IX'	The variable PRESSURE is assigned the AI attribute value of the process object P_METER with index 22.
OBJ = LIST(LN = "P_METER", IX = 22)	Same as above but using attributes.
PRESSURE = 'OBJ.LN':P'OBJ.IX'	

## 7.5

## Predefined picture variables

There are some picture variables with predefined names and meanings. These variables can be used in pictures throughout the system, but their values depend on the circumstances in which they are used. The predefined picture variable names are:

VIDEO_NR	The logical monitor number of the current monitor. The variable is the same as MONn:BAN. It can gain integer values in the range 1 ... 100. The monitor number is selected when an application session is started. This variable should not be changed manually.
PIC_NAME	The name of the picture (main picture) displayed on screen at the moment. The variable can have text values containing a max. of 10 characters. This variable should not be changed manually.
CURSOR_POS	<p>This variable contains the coordinates of the cursor position of the last function key selection. It is always updated when a function key is selected, except when the system is in input mode (after the !INPUT_VAR, !INPUT_POS or !INPUT_KEY commands). The variable is used, for example, when building line command keys, see <a href="#">Section 8</a>.</p> <p>The variable is a vector with four elements:</p> $(x, y, x\_rel, y\_rel)$ <p>where</p> <p>x, y = the coordinates relative to the upper left corner of the picture, which is (1,1). x = 1 ... 80, y = 1 ... 48.</p> <p>x_rel, y_rel = coordinates related to the upper left corner of the window where the cursor is situated, which is = (1,1).</p>
KEY_POS	<p>This variable contains the coordinates of the last function key selection related to the upper left corner of the key. The variable is updated each time a function key is selected. The variable is a vector of two elements:</p> $(x\_rel, y\_rel)$ <p>where</p> <p>x_rel, y_rel = coordinates relative to the upper left corner of the key, which is (1,1).</p>
ENTER_POS	<p>This variable contains the coordinates of the last click on an ENTER key. The variable is updated each time a function key containing an !ENTER command which terminates !INPUT_VAR is pressed, no matter whether the ENTER key contains other statements or not. The variable is a vector of four elements:</p> $(x, y, x\_rel, y\_rel)$ <p>where</p> <p>x, y = coordinates related to the upper left corner of the picture (=1,1).</p> <p>x_rel, y_rel = coordinates related to the upper left corner of the window in where the ENTER key is situated (= 1,1).</p>

## 7.5.1 Examples

Statement	Meaning
#EXEC EVENT_>'VIDEO_NR':E #ON EVENT_>'VIDEO_NR':E !SHOW ....	An event object containing the video number is activated. In combination with #ON sequences, likewise defined with video number, the event is directed to the same monitor.
#IF %CURSOR_POS(2) = 40 #THEN .....	A statement is executed on the condition that the pressed line is number 40.
#PRINT 1 'PIC_NAME'	The current screen picture is printed.



# Section 8      Expressions

This section describes how to compose SCIL expressions using various types of operands and operators. Special purpose context sensitive expressions defined by SCIL Data Derivation Language are also described.

<a href="#">Section 8.1</a>	<a href="#">General principles</a>
<a href="#">Section 8.2</a>	<a href="#">Arithmetical operators</a>
<a href="#">Section 8.3</a>	<a href="#">Relational operators</a>
<a href="#">Section 8.4</a>	<a href="#">Logical operators</a>
<a href="#">Section 8.5</a>	<a href="#">SCIL Data Derivation Language</a> <a href="#">SCIL Data Derivation Language (SDDL)</a>

## 8.1      General principles

### 8.1.1    Use

In SCIL, expressions are used as follows:

- for value assignments (objects, attributes, variables)
- as arguments for functions and commands
- as operands in expressions

### 8.1.2    Composition

Expressions are composed of operands and operators (possibly enclosed in parentheses). The operators are symbols for operations (for example, + : addition). The operands constitute the objects of these operations.

### 8.1.3    Operands

An operand may be:

- a constant ([Section 5](#))
- a variable or a component of a variable ([Section 7](#))
- a system or application object attribute or its component ([Section 6](#))
- an attribute of a Visual SCIL or window object or its component ([Section 6](#))
- a function call ([Section 10](#))
- a vector or a list aggregate ([Section 5](#))
- a named program or a method call, provided that the program returns a value ([Section 6](#))
- an expression enclosed in parentheses

The simplest expression is one single operand.

The data types of the operands determine which operations may be carried out on them and what the data type of the result will be. The data compatibility rules for each operator are given in figures below. List type operands cannot be operated upon by any operator. The list data type is therefore omitted from the compatibility rule figures. The data type of an expression can be read with the function DATA\_TYPE (see [Section 10](#)).

## 8.1.4 Operators

There are three types of SCIL operators:

- arithmetical operators
- relational operators
- logical operators

In SCIL expressions, they are evaluated in the order of priority mentioned below.

The operators and their usage are described in the next sections.

## 8.2 Arithmetical operators

### 8.2.1 Use

Arithmetical operators are used for numerical calculations. As operands, they expect numeric values, except for addition, which accepts text, bit string and byte string operands as well ([Figure 10](#)).

### 8.2.2 Operators

SCIL has the following arithmetical operators:

+	addition, positive sign
-	subtraction, negative sign
*	multiplication
/	division
**	exponential operator
DIV	integer division, the remainder is truncated from the result
MOD	modulus operator (the remainder by integer division provided that the operators are positive).

The operators DIV and MOD must be enclosed by spaces. For the other operators, spaces are optional.

Sigmas are valid for numeric data types (integers, real numbers and vectors of integers and real numbers) only.

### 8.2.3 Priority order

The order of priority for arithmetical operators, i.e. the order in which different parts of an expression are evaluated, does not differ from that in mathematics. It is as follows (operators with the highest priority first):

- 1) \*\*
- 2) / , \* , DIV , MOD
- 3) + , -

Operations with the same order of priority are evaluated from left to right.

## 8.2.4 Compatibility rules

Below are rules for which data types can be combined by means of the arithmetical operators and the data type of the resulting value.

Addition, +:		Right op.						
		I	R	B	T	C	S	V
Left op.		I	-	-	-	T	-	V
I = Integer		I	R	-	-	T	-	V
R = Real		R	R	-	-	R	-	V
B = Boolean		-	-	-	-	-	-	-
T = Time		-	-	-	-	-	-	V
C = Text		-	-	-	-	C	-	V
S = Bit/Byte String		-	-	-	-	-	S	V
V = Vector		V	V	-	V	V	V	V
- = Not allowed								

Figure 10: Addition rules

Subtraction, -:		Right op.						
		I	R	B	T	C	S	V
Left op.		I	-	-	-	-	-	V
I = Integer		I	R	-	-	-	-	V
R = Real		R	R	-	-	-	-	V
B = Boolean		-	-	-	-	-	-	-
T = Time		-	-	-	I	-	-	V
C = Text		-	-	-	-	-	-	-
S = Bit String		-	-	-	-	-	-	-
V = Vector		V	V	-	V	-	-	V
- = Not allowed								

Figure 11: Subtraction rules

When adding a numeric (integer or a real) operand to a time operand, or subtracting it from a time operand, the numeric operand is considered as seconds and the result is a moment of time specified number of seconds in the future or in the past.

When two time type operands are subtracted, the result is an integer number stating the time difference in seconds.

The result of adding two texts, two bit strings or two byte strings is the concatenation of the two strings.

When one operand is a simple value and the other one is a vector, the operation is carried out element by element and the result is a vector.

When both operands are vectors, the operation is carried out element by element and the result is a vector of the length of the longer operand. If the vectors are different in length, the missing elements of the shorter vector are regarded as zero or an empty string (depending on the data type of the odd element) and the status of the result element is set to SUSPICIOUS\_STATUS.

Multiplication, *:		Right op.						
Left op.		I	R	B	T	C	S	V
I = Integer	I	I	R	-	-	-	-	V
R = Real	R	R	R	-	-	-	-	V
B = Boolean	B	-	-	-	-	-	-	-
T = Time	T	-	-	-	-	-	-	-
C = Text	C	-	-	-	-	-	-	-
S = Bit String	S	-	-	-	-	-	-	-
V = Vector	V	V	V	-	-	-	-	V
- = Not allowed								

Figure 12: Multiplication rules

Division, /:		Right op.						
Left op.		I	R	B	T	C	S	V
I = Integer	I	R	R	-	-	-	-	V
R = Real	R	R	R	-	-	-	-	V
B = Boolean	B	-	-	-	-	-	-	-
T = Time	T	-	-	-	-	-	-	-
C = Text	C	-	-	-	-	-	-	-
S = Bit String	S	-	-	-	-	-	-	-
V = Vector	V	V	V	-	-	-	-	-
- = Not allowed								

Figure 13: Division rules

Integer operators		Right op.						
Left op.		I	R	B	T	C	S	V
<b>DIV and MOD:</b>								
I = Integer	I	I	-	-	-	-	-	V
R = Real	R	-	-	-	-	-	-	-
B = Boolean	B	-	-	-	-	-	-	-
T = Time	T	-	-	-	-	-	-	-
C = Text	C	-	-	-	-	-	-	-
S = Bit String	S	-	-	-	-	-	-	-
V = Vector	V	V	-	-	-	-	-	-
- = Not allowed								

Figure 14: DIV and MOD rules

When one operand is a simple value and the other one is a vector, the operation is carried out element by element and the result is a vector.

When both operands are vectors (allowed only in multiplication), the operation is carried out element by element and the result is a vector of the length of the longer operand. If the vectors are different in length, the missing elements of the shorter vector are regarded as 1 and the status of the result element is set to SUSPICIOUS\_STATUS.

A vector cannot be divided by another vector.

Left op.	Right op.						
	I	R	B	T	C	S	V
I	R	R	-	-	-	-	-
R	R	R	-	-	-	-	-
B	-	-	-	-	-	-	-
T	-	-	-	-	-	-	-
C	-	-	-	-	-	-	-
S	-	-	-	-	-	-	-
V	-	-	-	-	-	-	-

*Figure 15: Exponent rules*

If both operands are integers and the right operand is non-negative, the result is an integer. For example,  $(-2)^{**3}$  evaluates to integer -8.

## 8.2.5 Examples

Below are some examples of arithmetical operations. The bit string operands are written here by means of the BIT\_SCAN function, which creates a bit string out of its text representation.

Expression	Result
"A" + "B"	"AB"
$3^{**2}$	9
$3.0^{**2}$	9.0
$5 \text{ DIV } 2$	2
$5 \text{ MOD } 2$	1
$2^3 + 4/2$	8.0
$(2^3 + 4)/2$	5.0
BIT_SCAN("0101") + BIT_SCAN("0011")	BIT_SCAN("01010011")

## 8.3 Relational operators

### 8.3.1 Use

Relational operators are used for comparing expressions. The result of a comparison is always a boolean value, i. e. the value of a relation is either TRUE or FALSE.

### 8.3.2 Operators

The following relational operators are available in SCIL:

$==$	equal to
$>$	greater than
$<$	less than
$\neq$	unequal
$\leq$	less than or equal to
$\geq$	greater than or equal to

The relational operators have a lower order of priority than the arithmetical operators. Accordingly, arithmetical expressions are evaluated before comparisons are carried out.

### 8.3.3 Compatibility rules

[Figure 16](#) presents rules for what data types may be compared by means of relational operators.

Relational operators:		Right op.						
		I	R	B	T	C	S	V
I = Integer R = Real B = Boolean T = Time C = Text S = Bit String V = Vector - = Not allowed	I	B	B	-	B	-	-	-
	R	B	B	-	B	-	-	-
	B	-	-	B	-	-	-	-
	T	B	B	-	B	-	-	-
	C	-	-	-	-	B	-	-
	S	-	-	-	-	-	B	-
	V	-	-	-	-	-	-	-

*Figure 16: Rules for relational operators*

Comparison of text expressions is based on the Unicode presentation of each character, a number in the range 0 ... 65 535. The character codes are ordered so, that the digits (0 .. 9) precede the letters, the letters A - Z are arranged in alphabetical order and the uppercase letters A - Z precede the lower-case letters a - z. Comparison does not follow the current locale.

Boolean values and text may be compared only with values of the same data type. Time data can be compared to time data, integer and real values.

Bit strings are compared bit by bit starting from the leftmost bit, which is the most significant bit. If two bit strings of different length are identical to their common length, the shorter one is considered smaller.

Byte strings are compared numerically byte by byte (each byte has a value 0 ... 255), the first byte being the most significant. If two byte strings of different length are identical to their common length, the shorter one is considered smaller.

Vectors cannot be compared by relational operators.

### 8.3.4 Examples

Below are some examples of expressions containing relational operators.

Expression	Result
"B" > "A"	TRUE
%A - 30 == 0	TRUE, if %A == 30
OBJ:PRT + 60 >= CLOCK	TRUE, as long as less than one minute has passed since the latest registration time
BIT_SCAN("0011") < BIT_SCAN("0101")	TRUE
BIT_SCAN("0101") < BIT_SCAN("010101")	TRUE
BIT_SCAN("000111") < BIT_SCAN("0101")	TRUE
BIT_SCAN("010100") > BIT_SCAN("0101")	TRUE

## 8.4 Logical operators

### 8.4.1 Use

With logical operators, boolean values can be operated upon.

### 8.4.2 Operators

In SCIL there are the following logical operators:

AND	conjunction, "both .. and"
OR	disjunction, "or"
XOR	exclusive or, "either, but not both"
NOT	logical negation, the opposite

Logical operators have a lower order of priority than the relational operators, that is, relational expressions are evaluated before the logical operations are carried out. Logical operator NOT takes precedence of the other logical operators.

If an expression contains two or more different logical operators from the set (AND, OR, XOR), parentheses are required to explicitly specify the order. See the example below.

### 8.4.3 Compatibility rules

All logical operators expect operands of boolean type. Likewise, the results of logical operations are of boolean type.

### 8.4.4 Examples

Imagine that A and B are boolean data with the values shown to the left. The logical operators give the following results:

A	B	A AND B	A OR B	A XOR B	NOT A
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Wrong:                    A == 1 AND B == 2 OR C == 3  
 Right:                    (A == 1 AND B == 2) OR C == 3  
                           or                    A == 1 AND (B == 2 OR C == 3)

## 8.5 SCIL Data Derivation Language (SDDL)

### 8.5.1 General

SCIL Data Derivation Language (SDDL) is a language used to describe a data value calculated from context variables by using predefined SCIL operators and functions. The rule for calculating the data value is given as an SDDL expression.

The values of context variables are defined by the context where the value is evaluated. Typically, the context is a SYS600 object and context variables are attributes of the object. Some other context variables may be defined by the use case, for example APL and SYS, which represent the application and system object containing the context.

The language is designed in a way that allows for the calculation of the data value in a predictable time and deep in the database within the database lock. Consequently, an SDDL expression can be calculated, for example, as a part of the atomic transaction generated by an external event.

## 8.5.2 Relation to SCIL language

Syntactically, an SDDL expression is similar to an expression of SCIL. Context variables are used in the same way as local variables are used in SCIL.

An SDDL expression is not allowed to access any data outside its context. Therefore, it may not contain any direct references to MicroSCADA objects (such as MY\_OBJECT:POV1) nor references to SCIL program components (such as global variables or Visual SCIL objects). Furthermore, it may not call any SCIL function that accesses data outside the context.

Variable expansions are not allowed in an SDDL expression.

## 8.5.3 Context variables

Context variables share the name and value of the attributes of the context object.

In addition to the attributes of the context object, the following predefined context variables are declared:

APL	The base system object (APL:B) of the containing application.
SYS	The base system object SYS:B
TYPE	The object type of the context ("P", "IX", "X", "H", "F", "D", "C", "T", "A" or "G")
SUBTYPE	The subtype of the object, for example "BI", "AI", "NOD".

Context variables TYPE and SUBTYPE are available in any SDDL expression. Visibility of variables APL and SYS depends on the use case.

## 8.5.4 SDDL expression properties

An SDDL expression is deterministic, if it always evaluates to the same value when the values of context variables are the same. For example, a deterministic SDDL expression may not contain a RANDOM function call. As another example, an SDDL expression containing an HOUR function call (without arguments) is not deterministic, but HOUR(RT) is a deterministic SDDL expression.

An SDDL expression is static, if its context variables refer only to the configuration attributes of the context object, otherwise it is dynamic.

An SDDL expression may not contain any function call that has or may have side effects. Good examples of functions with possible side effects are OPS\_CALL and DO.

## 8.5.5 Examples

### Example 1

**SDDL expression**

```
APL.NA + ":" + LN + "." + dec(IX, 0)
```

defines a string data, which combines the application name, process object name and index to one string. APL, LN and IX are context variables.

This SDDL expression could be used as the NP (Name Pattern) attribute of a logging profile object, see the Application Objects manual.

**Example 2****SDDL expression**

```
(IL.BAY == "MyBay") and (IU == 1)
```

defines a boolean data, which tells whether the context object belongs to bay MyBay and is in use, or not.

**Example 3****SDDL expression**

```
if(OS == 0, "Good", "Not so good")
```

results in text "Good", if the status of the context object is 0, otherwise in text "Not so good".

**Example 4****SDDL expression**

```
join(".", IE)
```

presents the object identifier of the object as dot separated fields.



# Section 9 SCIL statements

This section describes the SCIL statements. It is divided into four sections as follows:

<a href="#">Section 9.1</a>	<a href="#">General</a> : The different types of SCIL statements, their arguments, and a table over all general SCIL statements, Visual SCIL commands and picture commands.
<a href="#">Section 9.2</a>	<a href="#">General SCIL statements</a> : The statements listed and described.
<a href="#">Section 9.3</a>	<a href="#">Visual SCIL commands</a> : The Visual SCIL commands listed and described.
<a href="#">Section 9.4</a>	<a href="#">Picture handling commands</a> : The picture handling commands listed and described.

## 9.1 General

### 9.1.1 Types of SCIL statements

There are five main types of SCIL statements (imperative, one-line statements are normally called commands):

- General SCIL statements. These are the basic SCIL statements that control the program flow, assign values to variables, access various system components such as system and application objects and files, etc. General SCIL statements are characterized by a starting # (number sign) and they can be used in all types of SCIL programs.
- Visual SCIL Commands. These commands operate on user interface objects (Visual SCIL objects, windows and picture functions) and often affect the visual appearance of the screen. Visual SCIL commands are characterized by a starting period (.) and they can be used in dialogs and pictures displayed in a VS (Visual SCIL) monitor.
- Picture commands. These commands operate on various components of pictures: windows, picture functions, function keys, etc. Picture commands are characterized by a starting ! (exclamation mark) and allowed only in pictures.
- Primitive graphics commands. These commands are used to draw graphical elements, such as lines, circles and text, on the screen. Primitive graphics commands can be used in pictures and Visual SCIL objects. The graphics commands are described in [Section 11](#).

### 9.1.2 Arguments

Most commands require arguments to become complete statements. The arguments specify the command with operands or key words. In this section, the commands are written in uppercase letters and the arguments in lowercase letters. Arguments in square brackets [] are optional. Argument enclosed by []\* may be repeated a number of times or it may be omitted. There must be at least one space character between a command and its arguments.

## 9.1.3 Overview

Table 2: General SCIL statements and Visual SCIL commands.

Statement	Brief Description	Page
Assignment statement	Assigns a value to a variable.	<a href="#">Section 9.2.1.1</a>
#ARGUMENT	Declares arguments of the program.	<a href="#">Section 9.2.1.2</a>
#BLOCK , #BLOCK_END	Compounds statements into one.	<a href="#">Section 9.2.1.3</a>
#CASE, #CASE_END	Multibranched conditional execution.	<a href="#">Section 9.2.1.4</a>
#CLOSE_FILE	Closes a keyed file.	<a href="#">Section 9.2.5.1</a>
#CREATE	Creates a new application or base system object.	<a href="#">Section 9.2.2.1</a>
#CREATE_FILE	Creates and opens a new keyed file.	<a href="#">Section 9.2.5.2</a>
#DELETE	Deletes an application object.	<a href="#">Section 9.2.2.2</a>
#DELETE_FILE	Deletes a file.	<a href="#">Section 9.2.5.3</a>
#DO	Executes the SCIL program given as an argument.	<a href="#">Section 9.2.1.5</a>
#ELSE	Conditional execution.	<a href="#">Section 9.2.1.8</a>
#ELSE_IF	Conditional execution.	<a href="#">Section 9.2.1.8</a>
#ERROR CONTINUE, #ERROR EVENT #ERROR IGNORE, #ERROR STOP	Defines the error handling policy.	<a href="#">Section 9.2.1.6</a>
#ERROR RAISE	Raises a SCIL error.	<a href="#">Section 9.2.1.7</a>
#EXEC	Queues an application object for execution.	<a href="#">Section 9.2.2.3</a>
#EXEC_AFTER	Queues an application object for execution after a time delay.	<a href="#">Section 9.2.2.4</a>
#GET	Updates process object values.	<a href="#">Section 9.2.2.5</a>
#IF	Conditional execution.	<a href="#">Section 9.2.1.8</a>
#INIT_QUERY	Initiates a process query.	<a href="#">Section 9.2.2.6</a>
#LIST	Prints process object data.	<a href="#">Section 9.2.3.1</a>
#LOCAL	Declares local variables of the program.	<a href="#">Section 9.2.1.9</a>
#LOOP, #LOOP_END	Executes a sequence of statements in a loop.	<a href="#">Section 9.2.1.10</a>
#LOOP_EXIT	Interrupts a loop.	<a href="#">Section 9.2.1.12</a>
#LOOP_WITH, #LOOP_END	Executes a loop a number of times.	<a href="#">Section 9.2.1.11</a>
#MODIFY	Changes an application object definition.	<a href="#">Section 9.2.2.7</a>
#ON	Declares a program block to be executed when an event occurs.	<a href="#">Section 9.2.1.13</a>
#ON ERROR	Defines an error handler.	<a href="#">Section 9.2.1.14</a>
#ON KEY_ERROR	Defines a key error handler.	<a href="#">Section 9.2.1.15</a>
#OPEN_FILE	Opens a keyed file.	<a href="#">Section 9.2.5.4</a>
#OTHERWISE	Multibranched conditional execution.	<a href="#">Section 9.2.1.4</a>
#PATH	Defines a logical path.	<a href="#">Section 9.2.4.1</a>
#PAUSE	Takes a pause.	<a href="#">Section 9.2.1.16</a>
#PRINT	Prints a picture.	<a href="#">Section 9.2.3.2</a>
#READ	Reads a data record from a keyed file.	<a href="#">Section 9.2.5.5</a>
#READ_KEYS	Reads the keys of a keyed file.	<a href="#">Section 9.2.5.6</a>
#READ_NEXT	Reads a data record from a keyed file.	<a href="#">Section 9.2.5.7</a>
#READ_PREV	Reads a data record from a keyed file.	<a href="#">Section 9.2.5.8</a>
#REMOVE	Deletes a data record from a keyed file.	<a href="#">Section 9.2.5.9</a>

Table continues on next page

<b>Statement</b>	<b>Brief Description</b>	<b>Page</b>
#RENAME_FILE	Renames a file.	<a href="#">Section 9.2.5.10</a>
#REP_LIB	Defines a logical representation library.	<a href="#">Section 9.2.4.2</a>
#RETURN	Stops the program execution and returns a value to the caller.	<a href="#">Section 9.2.1.17</a>
#SEARCH	Initialises a search among objects.	<a href="#">Section 9.2.2.8</a>
#SET	Assigns a value to an attribute of an object.	<a href="#">Section 9.2.2.9</a>
#SET_TIME	Sets the system time.	<a href="#">Section 9.2.1.18</a>
#WHEN	Multibranched conditional execution.	<a href="#">Section 9.2.1.4</a>
#WRITE	Writes a data record into a keyed file.	<a href="#">Section 9.2.5.11</a>
Method call	Calls a method.	<a href="#">Section 9.3.2.1</a>
.CREATE	Creates a Visual SCIL object.	<a href="#">Section 9.3.1.1</a>
.DELETE	Deletes a Visual SCIL object.	<a href="#">Section 9.3.1.2</a>
.LOAD	Loads a Visual SCIL object.	<a href="#">Section 9.3.1.3</a>
.MODIFY	Modifies one or more attributes of a Visual SCIL object.	<a href="#">Section 9.3.2.2</a>
.SET	Assigns a value to a user interface object attribute.	<a href="#">Section 9.3.2.3</a>

*Table 3: Picture commands*

<b>Command</b>	<b>Brief Description</b>	<b>Page</b>
!CLOSE	Closes the monitor.	<a href="#">Section 9.4.1.1</a>
!CSR_BOL, !CSR_EOL, !CSR_LEFT, !CSR_RIGHT	These commands move the data entry cursor.	<a href="#">Section 9.4.3.1</a>
!ENTER	Completes data entry.	<a href="#">Section 9.4.3.2</a>
!ERASE	Erases a window from the screen.	<a href="#">Section 9.4.2.1</a>
!FAST_PIC	Adds and removes fast picture definitions in semi-graphic monitors.	<a href="#">Section 9.4.1.2</a>
!INPUT_KEY	Reads function key information.	<a href="#">Section 9.4.3.2</a>
!INPUT_POS	Reads the mouse or cursor position.	<a href="#">Section 9.4.3.3</a>
!INPUT_VAR	Reads an input value from the user.	<a href="#">Section 9.4.3.4</a>
!INT_PIC	Displays an alarm picture.	<a href="#">Section 9.4.1.3</a>
!LAST_PIC	Displays the previous picture.	<a href="#">Section 9.4.1.4</a>
!NEW_PIC	Displays a picture.	<a href="#">Section 9.4.1.5</a>
!RECALL_PIC	Recalls a stored picture name.	<a href="#">Section 9.4.1.6</a>
!RESET	Deletes variables in a picture.	<a href="#">Section 9.4.4.2</a>
!RESTORE	Stops the function key from blinking.	<a href="#">Section 9.4.1.7</a>
!RUBOUT, !RUBOUT_BOL, !RUBOUT_CUR, !RUBOUT_EOL	Delete input data.	<a href="#">Section 9.4.3.5</a>
!SEND_PIC	Copies the picture (semi-graphic) to a printer.	<a href="#">Section 9.4.4.1</a>
!SHOW	Shows a window.	<a href="#">Section 9.4.2.2</a>
!SHOW_BACK	Shows the picture background of a window.	<a href="#">Section 9.4.2.3</a>
!STORE_PIC	Stores the present picture name.	<a href="#">Section 9.4.1.8</a>
!TOGGLE_MOD	Insert/typeover.	<a href="#">Section 9.4.3.5</a>
!UPDATE	Defines the update time interval.	<a href="#">Section 9.4.1.9</a>
!WIN_BG_COLOR	Specifies the color of the background behind the window.	<a href="#">Section 9.4.2.4</a>
!WIN_CREATE	Creates a window.	<a href="#">Section 9.4.2.5</a>

Table continues on next page

Command	Brief Description	Page
!WIN_INPUT	Assigns a window an expression.	<a href="#">Section 9.4.2.6</a>
!WIN_LEVEL	Specifies the level parameter of the window.	<a href="#">Section 9.4.2.7</a>
!WIN_NAME	Creates a new window.	<a href="#">Section 9.4.2.8</a>
!WIN_PIC	Selects a picture to be shown in the window.	<a href="#">Section 9.4.2.9</a>
!WIN_POS	Positions a window.	<a href="#">Section 9.4.2.10</a>
!WIN_REP	Selects a library representation for a window.	<a href="#">Section 9.4.2.11</a>

## 9.2 General SCIL statements

### 9.2.1 Basic SCIL statements

These statements define the structure and hence, the flow of the program. They are also used to declare arguments and local variables of the program and to assign values to variables.

Structured SCIL statements (#BLOCK, #CASE, #IF, #LOOP, #LOOP\_WITH, #ON and a method call) may be nested to depth of 300. For example, a loop may contain a block, which contains another loop that calls a method, which contains a case statement, etc., to the structural depth of 300 statements.

#### 9.2.1.1 **[@]name[component]\* = value**

Assigns a value to a variable.

'name'	The name of the variable
'component'	Component of structured data, see <a href="#">Section 5</a>
'value'	Any type expression

This statement replaces the current value of a variable or a component of a variable by a new value. With the @ prefix, 'name' refers to a global variable, otherwise a local variable is referred to.

Examples:

```
@X = Y ;Local variable Y is copied to global variable X
Y = %X ;Global variable X is copied to local variable Y
@X(5) = 1 ;The 5th element of global vector variable X is set
Y.A = 0 ;Attribute A of (list type) local variable Y is
Y.A.B(1 .. 5) = cleared
0 ;The 5 first elements of vector attribute B of
;list attribute A of local variable Y are cleared
```

#### 9.2.1.2 **#ARGUMENT name [,name]\***

Declares arguments of the program.

'name'	Any valid SCIL name, the name of the argument.
--------	--

The #ARGUMENT statement names the arguments to be passed to the program by the caller (in the order they should be given by the caller). All arguments may be listed in one statement, or several subsequent #ARGUMENT statements may be written. The two ways are equivalent as long as the arguments are declared in same order.

Arguments may be freely named using up to 63-character long identifiers. Arguments and global variables may have a same name. If an argument has a name of a predefined SCIL language element, such as a SCIL function, the predefined meaning is hidden and it cannot be used within that SCIL program. For example, if a SCIL program declares an argument named MAX, the predefined SCIL function MAX is no more available in the program.

The #ARGUMENT statements must be located at the beginning of program, before any other statements. Within the program, the argument name 'name' may be used as if it were a read-only local variable.

When a SCIL program that has declared its arguments is called by another SCIL program, the number of arguments supplied by the caller is checked. If the caller does not supply a value to each named argument, a SCIL error SCIL\_ACTUAL\_ARGUMENT\_MISSING is raised. On the other hand, additional arguments are allowed. They may be handled by the called program using SCIL functions ARGUMENT\_COUNT and ARGUMENT, see the example below.

**Example:**

This example shows how a method with two obligatory and one optional argument may be implemented.

```
; Calling sequence of this method:  
; MY_METHOD(A, B [,C])  
; Default value for the optional argument C is 0.  
  
#ARGUMENT A, B  
#LOCAL C = 0  
#IF ARGUMENT_COUNT == 3 #THEN C = ARGUMENT(3)
```

### 9.2.1.3 #BLOCK[statement]\* #BLOCK\_END

Compounds statements into one.

'statement'	Any SCIL statements.
-------------	----------------------

#BLOCK compounds a number of statements into one statement. It is usually used within structured statements (IF, CASE, ON) in places where a single statement is required by the syntax.

**Example:**

```
#IF A > B #THEN #BLOCK  
  #IF A - B > 10 #THEN #BLOCK  
    !SHOW WIN "THE VALUE TOO LARGE"  
    TOO_LARGE = TRUE  
  #BLOCK_END  
  #ELSE !SHOW WIN "CHECK!"  
#BLOCK END  
#ELSE !SHOW WIN "OK"
```

### 9.2.1.4 #CASE value, [ #WHEN selector statement]\*, [ #OTHERWISE statement], #CASE\_END

Multibranched conditional execution.

The statement selects (at most) one of listed SCIL statements for execution. The selection is based on a case value, which is compared to 'selectors' of each branch.

'value'	A value of type integer, real, boolean, time, text or bit string.
'selector'	The selector is a comma-separated list of items that select the branch for execution. An item may be given as:
	<ol style="list-style-type: none"> <li>1. A single value of type integer, real, boolean, time, text or bit string.</li> <li>2. A vector of such values. In this case 'value' is compared to each element of the vector to find a match.</li> <li>3. A range of such values, v1 .. v2. The range may be semi-open:</li> </ol>
	<p>v1 .. means values greater or equal to v1</p> <p>.. v2 means values less or equal to v2.</p>

'statement' Any SCIL statement.

The 'value' is compared to each 'selector' item of the #WHEN commands. The first matching #WHEN statement is executed and the rest of the #CASE statement to the matching #CASE\_END is skipped. If no matching #WHEN statement is found, the #OTHERWISE statement (if any) is executed.

There may be several selectors that match the case value, but only the first branch is executed. No error is generated, in case no branch is selected.

Examples:

```
#CASE C
#WHEN "A".."Z", "a" .. "z" #BLOCK
    IS_A LETTER = TRUE
    #CASE C
        #WHEN "A", "E", "I", "O", "U", "Y", -
            "a", "e", "i", "o", "u", "y" IS_A VOWEL = TRUE
        #OTHERWISE IS_A VOWEL = FALSE
    #CASE_END
#BLOCK_END
#OTHERWISE IS_A LETTER = FALSE
#CASE_END

#CASE D
#WHEN 0,3,6,8,9 SHAPE = "NO STRAIGHT LINES IN THIS DIGIT"
#WHEN 1,2,4..5,7 SHAPE = "CONTAINS STRAIGHT LINES"
#WHEN .. -1 #BLOCK
    WISE GUY = TRUE
    SHAPE = "DON'T TRY TO FOOL ME"
#BLOCK_END
#OTHERWISE SHAPE = "ONE DIGIT ONLY, PLEASE"
#CASE_END

#CASE DATA_TYPE(D)
#WHEN "INTEGER" T = 1
#WHEN "REAL" T = 2
#OTHERWISE !SHOW W "SIMPLE NUMERIC DATA EXPECTED"
#CASE_END
```

### 9.2.1.5 #DO program

Executes the SCIL program given as an argument.

'program'	A text or a text vector containing the statement or the program to be executed.
-----------	---

The #DO command is used to execute a SCIL program stored outside the current program context (picture, dialog, command procedure) or created on-the-fly.

The program is executed as a subroutine of the calling program, contrary to the #EXEC command, which queues the specified object for later execution.

#DO command cannot pass arguments to the called program, nor does it support return values from the called program. Use function DO instead.

**Example:**

```
#LOCAL PROG = TEXT_READ("ABC.TXT")
#IF PROG.STATUS == 0 #THEN #DO PROG.TEXT
;The SCIL program in the file ABC.TXT is executed.

#DO ABC:C
;The program (IN attribute) of the command procedure ABC is executed.
```

### 9.2.1.6 #ERROR IGNORE, #ERROR CONTINUE, #ERROR STOP, #ERROR EVENT

Defines the error handling policy.

IGNORE	means that the program execution continues regardless of errors. The error handling programs are not activated and no error message is produced.
CONTINUE	means that an error handler is activated or an error message is produced, but the program execution continues.
STOP	means that an error handler is activated or an error message is produced and the execution of the program containing the error is aborted. The statement of an active #ON ERROR command is executed (see below).
EVENT	means that the execution of the program containing the error is aborted, but no error message is produced. The statement of an active #ON ERROR command is executed (see below). Not available in the methods of dialogs.

An #ERROR command applies only to the program or #ON block in which it is executed.

The status code of the most recent error occurred in the program can be read with the STATUS function, [Section 10](#).

If no ERROR command has been executed in a program, the following default policies are applied:

Background and draw programs:	CONTINUE
Start programs:	STOP when displayed, IGNORE when printed
Update programs:	IGNORE
Exit Programs:	IGNORE
Key programs:	STOP
Named programs:	STOP
Command procedures:	STOP
#ON blocks:	IGNORE
Methods of VS objects:	STOP (IGNORE in the delete method)
Error handling programs:	IGNORE

In error handling programs, the error handling policy is always IGNORE, regardless of possible #ERROR commands. In delete methods of Visual SCIL objects all errors are ignored.

### 9.2.1.7 #ERROR RAISE [status]

Raises a SCIL error.

'status'

Integer expression. The status code to be activated. Default: the latest error code occurred in the program.

The command is mainly used within error handling blocks (#ON ERROR or #ON KEY\_ERROR blocks) in order to activate an error status in the program. In error handling blocks, 'status' can be omitted, whereby the activated status is the most recent error status that has occurred in the program. Outside an error handling block, the command interrupts the program execution.

Example:

```
#ON ERROR #BLOCK
  #IF STATUS == STATUS_CODE("SCIL_APPL_COMMUNICATION_TIMEOUT") #THEN -
    !SHOW INFO "Timeout"
  #ELSE #ERROR RAISE
#BLOCK_END
```

### 9.2.1.8 #IF condition1 #THEN statement1, [ #ELSE\_IF condition2 #THEN statement2]\*, [ #ELSE statement3]

Conditional execution.

'condition1'	
'condition2'	Boolean type expressions
'statement1'	
'statement2'	
'statement3'	Any SCIL statements.

If 'condition1' is TRUE, 'statement1' is executed and the rest of the #IF statement is skipped. Otherwise, if any 'condition2' is TRUE, the corresponding 'statement2' is executed and the rest is skipped. If none of 'condition1' or 'condition2' is TRUE, 'statement3' is executed.

Examples:

```
#LOCAL HOUR, SHIFT
#IF HOUR >= 7 AND HOUR < 15 #THEN SHIFT = "MORNING"
#ELSE_IF HOUR >= 15 AND HOUR < 23 #THEN SHIFT = "EVENING"
#ELSE SHIFT = "NIGHT"
```

### 9.2.1.9 #LOCAL name [= value] [,name [= value]]\*

Declares local variables of the program.

'name'	Any valid SCIL name, the name of the local variable.
'value'	Any type expression, the initial value of the variable.

The #LOCAL statement names the local variables to be used in the program. All local variables may be listed in one statement, or several subsequent #LOCAL statements may be written. The two ways are equivalent. Variables may be declared in any order.

Local variables may be freely named using up to 63-character long identifiers. Local and global variables may have a same name. If an argument has a name of a predefined SCIL language element, such as a SCIL function, the predefined meaning is hidden and it cannot be used within that SCIL program. For example, if a SCIL program declares a local variable named MAX, the predefined SCIL function MAX is no more available in the program.

The name of local variable may not be formed by using variable expansion. The following is not valid:

```
#LOCAL 'LN'_MAX
```

The #LOCAL statements must be located at the beginning of program, after #ARGUMENT statements (if any) but before any other statements.

Within the SCIL program, local variables are referred to simply by their name. No special characters (such as @ and % for global variables) are needed. Syntactically, local variables may be used wherever a global variable reference is allowed.

Additionally, in every SCIL command that takes a variable name as its parameter, a local variable name may be used (but not an argument name, because arguments are read-only). The list of such SCIL commands follows:

```
#LOOP_WITH var = low .. high
#OPEN_FILE n apl file keylength
#READ n key data1 [data2]
#READ_KEYS n vector [key1 [key2]]
#READ_NEXT n key data1 [data2]
#READ_PREV n key data1 [data2]
.MOUSE x, y, [,button [,buttons [,RELATIVE]]]
!INPUT_KEY keytext var
!INPUT_POS var
!INPUT_VAR [picture path]window variable [max_length]
```

The SCIL interpreter first finds out whether a local variable by the given name exists. If it does, the local variable is used, if not, a global variable by the name is used.

The local variables exist only while the program is executed. When the program terminates, all its local variables are destroyed and the memory space allocated for them is freed.

**Example:**

This example illustrates the use of local variables and arguments (which may be taken as read-only local variables).

```
#ARGUMENT A, B, LN
#LOCAL X, Y = A + B           ;Initial value of Y is the sum of arguments A and B
#LOCAL I
#LOCAL V = VECTOR()

X = %X                         ;Global X is copied to local X
X = MIN(X, Y)

#SET 'LN':PBO1 = 1            ;Argument LN is expanded to object name

#LOOP_WITH I = 1 .. 10          ;Local I used as a loop counter
    V(I) = .CALCULATE_SOMETHING(I, X, Y)
#LOOP_END
```

### 9.2.1.10 #LOOP [condition], [statement]\*, #LOOP\_END [max]

Executes a sequence of statements in a loop.

'condition'	A boolean expression, a precondition for entering the body of the loop. Default value: TRUE.
'statement'	Any SCIL statements, the body of the loop.
'max'	The maximum number of times the loop is iterated. An integer expression. Default value: 1000.

The body of the loop is executed repeatedly as long as the 'condition' is TRUE, or until the loop is interrupted in one of the following ways:

- Statement #LOOP\_EXIT (see below) is executed in the loop body. This is no error situation.
- The maximum number of loop iterations is reached. In this case, error SCIL\_MAX\_LOOP\_COUNT\_EXCEEDED is raised.
- An emergency interruption is done from another SCIL program (in another monitor). In this case error SCIL\_PROGRAM\_EXTERNALLY\_TERMINATED is raised.

An emergency interruption is done in one of the following ways:

- Loops in pictures and VS objects are interrupted by the statement:

```
#SET MONn:BMS
```

where 'n' is the monitor number.

- Loops in command procedures are interrupted by the statement:

```
#SET APILn:BRSm
```

where 'n' is the application number and 'm' represents the number of the queue that runs the command procedure. This number may be found by reading the APILn:BRO (Running Objects) attribute, it is encoded as follows:

'm' = 1: time channel queue

'm' = 2: event channel queue

'm' = 3 ... 32: parallel queue 1 ... 30 (the PQ attribute of the command procedure)

- Loops in format pictures are interrupted by the statement:

```
#SET APILn:BPSm
```

where

'n' is the application number,

'm' = 1 for process printouts and

'm' = 2 for report printouts.

#### Example:

```
#LOCAL I = 0, OBJ, NAMES = VECTOR()
#SEARCH 1 0 "P" "A"
OBJ = NEXT(1)
#LOOP OBJ.IU >= 0 AND I < 50
    I = I + 1
    NAMES(I) = OBJ.LN
    OBJ = NEXT(1)
#LOOP_END
!SHOW OBJECTS NAMES
;Up to 50 process object names are shown in the window OBJECTS.
```

### 9.2.1.11 #LOOP\_WITH var = low .. high, [statement]\*, #LOOP\_END

Executes a loop a number of times.

'var'	The name of the control variable of the loop. The name refers to a local variable by that name, if such a variable is declared, otherwise to a global variable.
'low'	The value of the control variable at the first loop execution, an integer expression.
'high'	The value of the control variable at the last loop execution, an integer expression.
'statement'	Any SCIL statements, the body of the loop.

The body of the loop is executed repeatedly a number of times, calculated as 'high' - 'low' + 1. Each time the loop is completed, the variable 'var' is incremented by one. If 'high' is less than 'low', the body is not executed at all.

The loop may be interrupted before 'var' reaching 'high' in the following ways:

- Statement #LOOP\_EXIT (see below) is executed in the loop body. This is no error situation.
- An emergency interruption is done from another SCIL program (in another monitor). In this case error SCIL\_PROGRAM\_EXTERNALLY\_TERMINATED is raised. See command #LOOP above for details of emergency interruption.

Example:

```
#LOOP_WITH I = 1 .. LENGTH(%V)
    !SHOW WIN'I' %V(%I)
#LOOP_END
;Each element of the vector variable V is shown in a separate window.
```

### 9.2.1.12 #LOOP\_EXIT

Interrupts a loop.

The statement interrupts the innermost loop (#LOOP or #LOOP\_WITH) it is textually located in.

Example:

```
I = 0
#LOOP
    I = I + 1
    !SHOW LOOP_NR I
    !SHOW QUESTION "CONTINUE? (Y/N)"
    !INPUT_VAR ANSWER ANSWER
    #IF ANSWER == "N" #THEN #LOOP_EXIT
#LOOP_END
```

### 9.2.1.13 #ON event [statement]

Declares a program block to be executed when an event occurs.

'event'	An event object notation.
'statement'	Any SCIL statement.

#ON command stores the 'statement' as the event program for 'event' in the current user interface object (main picture, window picture, picture function or Visual SCIL object). Later, the 'statement' will be executed each time the event object is activated. If 'statement' is omitted, the previous #ON command for the same event object is cancelled.

Only one event program for a particular event is stored in each user interface object, second #ON command for the same event replaces the first one. However, each window picture, picture function or VS object may have its own event program for each event.

This command can be used only in user interface programs. It has no effect in command procedures. Only events from the current application can be caught.

In Visual SCIL objects, it is recommended to define event programs as event methods of the object (instead of #ON blocks) using the Dialog Editor.

Examples:

```
#ON TEMP:E1 !SHOW W TEMP:PAI1
;When the event object TEMP:E1 is activated, the value of the process
object
;TEMP is shown in the window W.

#ON TEMP:E1
;The former statement is cancelled.

#ON SWITCH:E2 #BLOCK
    !SHOW SWITCH SWITCH:PBO2
    !SHOW TIME TIMES(SWITCH:PRT2)
#BLOCK_END
;The block is executed when the event object SWITCH:E2 is activated.
```

#### 9.2.1.14 #ON ERROR [statement]

Defines an error handler.

'statement'                    A SCIL statement to be executed when an error occurs.

The command defines a statement, or block of statements, to be executed each time an error occurs, in the cases where the error handling is defined by #ERROR STOP or #ERROR EVENT (see above). The command is valid only for the program or #ON block in which it has been executed. The #ON ERROR command takes precedence over the #ON KEY\_ERROR command (see below).

Example:

```
#ON ERROR !SHOW MESSAGE STATUS
;The window MESSAGE is shown when an error occurs.
```

#### 9.2.1.15 #ON KEY\_ERROR [statement]

Defines a key error handler.

'statement'                    A SCIL statement to be executed when an error occurs.

The command defines a statement, or a block of statements, to be executed in a picture each time an error occurs in a function key program. The command applies only to the window picture where it has been executed. If there is no key error handler in a picture, the error handler of its parent picture is used, if any. The #ON ERROR command (see above) takes precedence over the #ON KEY\_ERROR command.

Example:

```
#ON KEY_ERROR !SHOW MESSAGE "FUNCTION KEY ERROR"
```

### 9.2.1.16 #PAUSE interval

Takes a pause.

'interval' Time interval in seconds given as a real expression.

This command is used to momentarily suspend the program execution.



Be careful when using this command in command procedures! The pause delays the execution of all the other objects in the same queue as well.

## Examples:

```
#PAUSE 3.5
;The system waits for 3.5 seconds before the next statement is executed.
#PAUSE %T
;The length of the pause is specified by the variable T.
```

### 9.2.1.17 #RETURN [value]

Stops the program execution and returns a value to the caller.

'value' Any SCIL expression.

**#RETURN** command is used in named programs, methods and programs executed with the DO function (see [Section 10](#)) to stop execution and return a value to the caller.

The #RETURN command without 'value' may be used in any SCIL program to exit the program.

A program encountering no #RETURN statement returns a value with data type "NONE".

**Example:**

; Suppose you have the following named programs:

```
;Named program ADD:  
#ARGUMENT A, B  
#RETURN A + B
```

```
;Named program MULTI_ADD:  
#LOCAL I, TOTAL = 0  
#LOOP_WITH I = 1 .. ARGUMENT_COUNT  
    TOTAL = TOTAL + ARGUMENT(I)  
#LOOP_END  
#RETURN TOTAL
```

```
;After the following named program calls:  
S = .ADD(1, 0.5)  
M = .MULTI_ADD(1, 2, 3, 4)  
;S has the value 1.5 and M 10.
```

**9.2.1.18      #SET TIME time**

Sets the system time.

'time' Time given in the format YY-MM-DD HH:MM:SS (if SYS:BTF = 0), DD-MM-YY HH:MM:SS (if SYS:BTF = 1) or MM-DD-YY HH:MM:SS (if SYS:BTF = 2).  
The seconds may be omitted.

The command sets the time of the system clock. If the computer has an external clock, the command has no relevance.



This command is more or less obsolete. Use functions SET\_SYS\_TIME, SET\_LOCAL\_TIME or SET\_UTC\_TIME instead (see [Section 10](#)).  
This command is not allowed in read-only mode.

## 9.2.2 Application and system object commands

### 9.2.2.1 #CREATE object [=attributes]

Creates a new application or base system object.

'object' An object notation. Object types allowed are the application objects P, H, X, F, D, C, T, A, G and V, and the system objects B.  
'attributes' A list type expression.

The command creates a new object of the given type, with the given name (and index, if a process object) and assigns it the attribute values of 'attributes'.

If a process object notation for a process object of a predefined type is given without an index, a group with the given name is created. A group is automatically created when an indexed process object of a predefined type is created.

To make a copy of an existing object, use functions FETCH, PHYS\_FETCH, NEXT or PREV ([Section 10](#)).

An error is raised if the object already exists, or if the assigned 'attributes' do not match the actual object type. In read-only mode, only V objects can be created.

The command is mainly used in tool pictures and configuration programs.

**Example:**

```
#CREATE ABC:P1 = LIST(BI = 0, UN = 20, OA = 1, OB = 2)
;A binary input type process object is created with a physical address.
```

### 9.2.2.2 #DELETE object

Deletes an application object.

'object' An object notation of type P, H, X, F, D, C, T, A, G or V. A variable object notation (type V) may contain an attribute name.

The given object is deleted. If a variable object notation contains an attribute name, the attribute, not the entire object, is deleted. In read-only mode, only V objects and their attributes can be deleted.

Using a variable object notation, any global variable may be deleted, see the example below.

A process object group (process object notation without an index) may be deleted only if it does not contain any process objects. A time channel may be deleted only if no object is connected to it. A scale or an event handling object can be deleted only if no process objects use it. An free type object (F) can be deleted only if all process objects of the corresponding type have been deleted. A logging profile can be deleted only if no objects use it.

**Examples:**

```
@TMP = 1
#DELETE TMP:V
;The variable TMP is deleted

#DELETE A:P1
;The process object A with index 1 is deleted.

#DELETE A:P
;The process object group A is deleted (possible only if there are no
;objects ;in the group).

#DELETE V:VAB
;The attribute AB of the global variable V is deleted.
```

### 9.2.2.3 #EXEC object [(variable\_list)]

Queues an application object for execution.

'object'	An application object notation of type D, C, T, A or E.
'variable_list'	A list of variable assignments separated by commas.

This command queues an application object (data object, command procedure, time channel, event channel or event object) for execution. Because the object does not necessarily execute immediately, the subsequent statements should not assume that the object has been completed. See [Section 6.3](#) for the overview of application objects, or the Application Objects manual for details.

Any number of argument variables may be passed in 'variable\_list' to the activated object (except for event objects that do not take any arguments).

When executed by a command procedure, #EXEC command may fail by status REPF\_EXECUTION\_QUEUE\_FULL if the maximum queue length (attribute APL:BQM(2) or APL:BQM(3)) has been reached (see the System Objects manual).

This command is not available in read-only mode.

**Examples:**

```
#EXEC TASK:C (@LN = "DEFG", @IX = 1)
;The command procedure TASK is queued for execution.

#EXEC DATA:D (@A = %B + 2)
;The data object DATA is updated.

#EXEC EVENT:E1
;An event object named EVENT is activated.
```

### 9.2.2.4 #EXEC\_AFTER delay object [(variable\_list)]

Queues an application object for execution after a time delay.

'delay'	The delay time in seconds given as an integer or real expression.
'object'	An object notation of type D, C, T, A or E.
'variable_list'	A list of variable assignments separated by commas.

The command works like #EXEC (see above), but the execution is delayed for 'delay' seconds.

External applications are not supported. For example, the command #EXEC\_AFTER 60 EXTERNAL:5C fails if application 5 is an external application. As a work-around, create a local command procedure, say LOCAL, which contains only the command #EXEC EXTERNAL:5C and queue it for execution: #EXEC\_AFTER 60 LOCAL:C

#EXEC\_AFTER command may fail by status REPF\_EXECUTION\_QUEUE\_FULL if the maximum queue length (attribute APL:BQM(4)) has been reached (see the System Objects manual).

This command is not available in read-only mode.

Example:

```
#EXEC_AFTER 10 TASK:C (@LN = "DEFG", @IX = 1)
;The command procedure TASK is started after 10 seconds.
```

### 9.2.2.5 #GET object

Updates process object values.

'object'	A process object notation or an STA system object notation with the attribute ME or DA.
----------	---

The command reads the specified process object value(s) or a memory address or an address range from the station and updates the process database. For process objects, the attribute may be omitted. It is then assumed to be OV. The command is valid only for process objects corresponding to physical objects in stations on ANSI X.3 lines.

This command is not available in read-only mode.

Examples:

```
#GET A:PBI
#GET B:PBI(1 .. 4)
#GET STA1:SME(1003^ .. 1010^)
;The binary object A, the four first indices of the binary object B and
;the memory address area 1003 to 1010 from station 1 are updated in the
process
;database.
```

### 9.2.2.6 #INIT\_QUERY n [condition]

Initiates a process query.

'n'	Text expression, either "A", "P", "L", "H" or "E":
"A" (Alphabetical)	The entire process database is searched in alphabetical order.
"P" (Physical)	The entire process database is searched in the order of the physical addresses of objects (UN + OA + OB).

Table continues on next page

	"L" (alarm List)	The alarm list is searched in reverse time order, the latest alarms first. The alarm list contains all alarming and unacknowledged process objects in time order according to the alarm time (the AT and AM attributes).
	"H" (History)	The history buffer is searched in time order, the oldest events first. A history buffer is an obsolete way of storing event history, selected by the application attribute APL:BHP value "EVENT_LOG".
	"E" (Event)	The history buffer is searched in reverse time order, the latest events first. See "H" above.
'condition'		A boolean type expression which selects the objects to be included in the query. The condition is comprised of relations and logical operators. The relations have an attribute as the left operand. All attributes (including CA), except those of vector or list type, can be used. The objects that fulfil the condition are included in the query. Wildcard characters % and * can be used in conjunction with text attributes. % matches any character, * matches any sequence of characters (including a zero length sequence). The condition is evaluated in read-only mode.

This command only selects the objects included in the query. The objects along with some of their attribute values are then listed by function PROD\_QUERY ([Section 10](#)). Only one query at a time may be active within the SCIL context. Only own application may be queried.

Examples:

```
#INIT_QUERY "A"
;The process query includes all objects in alphabetical order.

#INIT_QUERY "P" UN == 5
;The objects of unit 5 in address order.

#INIT_QUERY "A" LN == "K*"
;Process objects beginning with K.

#INIT_QUERY "L" (UN == 5) AND (AR == 0)
;Unacknowledged alarms from unit 5.
```



This command is more or less obsolete. Use more powerful SCIL functions APPLICATION\_OBJECT\_LIST and APPLICATION\_OBJECT\_ATTRIBUTES instead of type "A" and "P" queries and function HISTORY\_DATABASE\_MANAGER (operating on the history database) instead of type "H" and "E" queries (operating on the history buffer).

### 9.2.2.7 #MODIFY object = attributes

Changes an application object definition.

'object'	An application object notation. Allowed object types are P, H, X, F, D, C, T, A, G and V.
'attributes'	A value of data type list.

The object is assigned the attribute values of the list expression 'attributes'. Modifying the LN attribute of an object (or the IX attribute of a process object) effectively renames the object.

In read-only mode, only V objects can be modified.

Examples:

```
#LOCAL V = LIST(HR = 24, TC = "TC_1H")
#MODIFY DEF:D = V
```

```
;The HR and TC attributes of the data object DEF are changed.

#MODIFY PROC:P2 = LIST(AE = 1,AN = "A_1")
;The process object PROC, index 2, is connected to the event channel A_1.

#MODIFY ABC:P2 = LIST(LN = "DEF")
;The process object is renamed, i.e. moved from group ABC to DEF.
```

### 9.2.2.8 #SEARCH n apl type order [start [condition]]

Initialises a search among objects.

'n'	The identification number of the search. An integer expression 1 ... 10 that identifies the search within the SCIL context.								
'apl'	Logical application number. Integer expression, 0 ... 250. 0 = the own application. The application must be local.								
'type'	The object type, "P", "H", "X", "F", "D", "C", "T", "A" or "G" given as a text expression.								
'order'	The search order given as a text expression: <table border="0"> <tr> <td>"A"</td><td>Alphabetical order. Searching through the object names in alphabetical order (no index). Regarding process objects, only group names are included in the search.</td></tr> <tr> <td>"I"</td><td>Index order (only for process objects of predefined types). Searching through the individual objects of a process object group.</td></tr> <tr> <td>"P"</td><td>Address order (only for process objects).</td></tr> <tr> <td>"E"</td><td>Execution order. The execution order within a time channel. 'type' can be either "D" or "C". Whichever given, the search still applies to both data objects and command procedures.</td></tr> </table>	"A"	Alphabetical order. Searching through the object names in alphabetical order (no index). Regarding process objects, only group names are included in the search.	"I"	Index order (only for process objects of predefined types). Searching through the individual objects of a process object group.	"P"	Address order (only for process objects).	"E"	Execution order. The execution order within a time channel. 'type' can be either "D" or "C". Whichever given, the search still applies to both data objects and command procedures.
"A"	Alphabetical order. Searching through the object names in alphabetical order (no index). Regarding process objects, only group names are included in the search.								
"I"	Index order (only for process objects of predefined types). Searching through the individual objects of a process object group.								
"P"	Address order (only for process objects).								
"E"	Execution order. The execution order within a time channel. 'type' can be either "D" or "C". Whichever given, the search still applies to both data objects and command procedures.								
'start'	The starting point of the search. Depending on 'order', the parameter has the following values: <table border="0"> <tr> <td>Order "A"</td><td>An object name as a text expression. If omitted, the search starts from the first name.</td></tr> <tr> <td>Order "I"</td><td>A name or (name,index), where 'name' is the name of the object given as a text expression and 'index' is the index of process object as an integer. If 'index' is omitted, the browsing starts from the first index.</td></tr> <tr> <td>Order "P"</td><td>A unit or (unit,address) or (unit, address, bit address), where 'unit' is the unit number, 'address' is the word address and 'bit address' is the bit address, each given as an integer expression.</td></tr> <tr> <td>Order "E"</td><td>A time channel or (time channel, type, name), where 'time channel' is the name of the time channel and 'type' and 'name' are the name and type of the connected object from which the search starts, each given as text expressions. If only the time channel name is given, the search starts from the object with the highest priority. Both data objects and command procedures are included.</td></tr> </table>	Order "A"	An object name as a text expression. If omitted, the search starts from the first name.	Order "I"	A name or (name,index), where 'name' is the name of the object given as a text expression and 'index' is the index of process object as an integer. If 'index' is omitted, the browsing starts from the first index.	Order "P"	A unit or (unit,address) or (unit, address, bit address), where 'unit' is the unit number, 'address' is the word address and 'bit address' is the bit address, each given as an integer expression.	Order "E"	A time channel or (time channel, type, name), where 'time channel' is the name of the time channel and 'type' and 'name' are the name and type of the connected object from which the search starts, each given as text expressions. If only the time channel name is given, the search starts from the object with the highest priority. Both data objects and command procedures are included.
Order "A"	An object name as a text expression. If omitted, the search starts from the first name.								
Order "I"	A name or (name,index), where 'name' is the name of the object given as a text expression and 'index' is the index of process object as an integer. If 'index' is omitted, the browsing starts from the first index.								
Order "P"	A unit or (unit,address) or (unit, address, bit address), where 'unit' is the unit number, 'address' is the word address and 'bit address' is the bit address, each given as an integer expression.								
Order "E"	A time channel or (time channel, type, name), where 'time channel' is the name of the time channel and 'type' and 'name' are the name and type of the connected object from which the search starts, each given as text expressions. If only the time channel name is given, the search starts from the object with the highest priority. Both data objects and command procedures are included.								

Table continues on next page

The argument 'start' is obligatory when 'order' is "I" or "P", and when a condition is appended to the statement. If 'order' is "A" or "E", 'start' can be given as an empty string ("").

Note, that the first NEXT or PREV function call after the #SEARCH command uses the 'start' argument as its reference: If an object specified by 'start' exists, that object is not returned by the call.

'condition'	A boolean type expression which selects the objects to be included in the browsing. The condition is comprised of relations and logical operators. The relations have an attribute as the left operand. All attributes, except vector or list type attributes, can be included in the expression. The objects that fulfill the condition are included in the search. Wildcard characters % and * can be used in conjunction with text attributes. % matches any character, * matches any sequence of characters (including a zero length sequence). The condition is evaluated in read-only mode.
-------------	--

This command only selects the objects included in the search. The objects along with their configuration attribute values are then listed by functions NEXT and PREV ([Section 10](#)). Up to 10 searches at a time may be active within the SCIL context.



This command is more or less obsolete. Use more powerful SCIL functions APPLICATION\_OBJECT\_LIST and APPLICATION\_OBJECT\_ATTRIBUTES instead.

Examples:

```
#SEARCH 1 0 "P" "A"
;All process objects in alphabetical order are included in search number 1.

#SEARCH 2 2 "T" "A"
;Browsing through all time channels in alphabetical order.

#SEARCH 3 1 "P" "P" (4, 1000^) LN == "B*"
;The search refers to those objects of station 4 which have an address
1000
;(octal number) and begin with B. The search is performed in address
order.

#SEARCH 4 0 "D" "E" "10MIN"
;Searching among those data objects and command procedures which are
connected
;to the time channel 10MIN.
```

### 9.2.2.9 #SET object\_attribute [= value]

Assigns a value to an attribute of an object.

'object_attribute'	An object notation of type P, H, X, F, D, C, T, A, G, B, S or V including the attribute to be set. See <a href="#">Section 6</a> .
'value'	A value of the type specified by the attribute. Default = 1.

With this command all types of system and application objects, except event objects, may be given values through their attributes. For real process objects of output type, setting the OV attribute implies control of the process via NET.

Depending on the object type and the attribute, the object notation may be indexed by a single index or an index range. If the object notation has an index range, the 'value' may be a vector or of a simple data type. If it is a vector, its elements list the values to be assigned. If it is simple data, all indices receive the same value.

Data object and process object notations may be used without an attribute. It is then assumed to be the OV attribute.

In read-only mode, only V objects can be set by this command. The attributes which are described as read-only in the attribute descriptions (Application Objects and System Objects manuals) cannot be set with the #SET command.

Examples:

```
#SET SWITCH:PBO1 = 1
;The switch is set to 1. If the object is in AUTO state, the command is
passed
;out to the process.

#SET STA3:SME(3121^0 .. 3121^7) = 0
;All storage bits in the given range are set to zero.

#SET TASK:CIU = 1
;The command procedure TASK is taken into use.

#SET DATA:DOV5 = 10.0
;The fifth history value of the data object is set to 10.0.

#SET A:AON = "B"
;The event channel A is connected to the object B.

#SET PRI1:BLP = 72
;The number of lines per page is set to 72 for printer 1.
```

## 9.2.3 Printout commands

### 9.2.3.1 #LIST printer object [(variable list)]

Prints process object data.

'printer'	Logical printer number. Integer expression, 1 ... 20.
'object'	A process object notation.
'variable list'	A list of variable assignments, separated by commas.

The format picture of the process object is output to the printer. If the process object is given with an index, the physical format picture (PF) is written, otherwise the logical format picture of the group (LF) is used. For process objects of user-defined types, the command always prints the physical format picture (PF).

The start program of the picture, exclusive of the picture commands, is executed before printout. Only those windows which have an expression are printed. Curves and bars are not printed. Depending on the printer definition in the base system (the PRIn:BOD attribute), the printout may be stored on disk.

The variable list, which may be omitted, defines the variables used in the printed picture (in windows or in start programs). The variable list may assign a value to a variable called FORM\_FEED, which determines the paper form feed, see the #PRINT command. In conjunction with #LIST command, the default value of FORM\_FEED is 0, which means that the printer starts a new page only when the previous one is full.

In addition to the variables of 'variable list', the #LIST command automatically defines a set of variables which get their names as well as their values from attributes (snapshot variables). If the process object is given with an index, these attributes may be (depending on the object type and definition): LN, IX, OV, BI, BO, DB, DI, DO, AI, AO, PC, BS, OE, AL, AS, OS, SE, SP, OF, AZ, RT, RM, AT, AM and CA. For user-defined object types also other attributes may also be

transferred to variables. The corresponding variables may be used in the physical format. If the process object is given without an index, only the LN attribute is passed in this way to the logical format. The variable values given in the variable list have precedence of these automatically generated variables.

This command is not available in read-only mode.

Examples:

```
#LIST 3 TEMP:P (@A = 30)
;The logical format of the process object group is printed to printer 3.
```

```
#LIST 1 LEVEL:P7 (@FORM_FEED = 2)
;The physical format of the process object is printed to printer 1. E.g.
;the
;variable LN has the value "LEVEL", and the variable IX the value 7. The
;printer starts a new page before, but not after the printout.
```

### 9.2.3.2 #PRINT printer picture [(variable list)]

Prints a picture.

'printer'	Logical printer number. Integer expression, 1 ... 20.
'picture'	The picture to be printed, specified as: [path/] picture name where 'picture name' is the name of the picture and 'path' is a logical path name. If 'path' is omitted, the default path names are used. See the #PATH command, <a href="#">Section 9.2.4</a> .
'variable list'	A list of variable assignments, separated by commas.

This command is used when a paper printout of a picture is needed, for example, a report picture. The start program of the picture, excluding the picture commands, is executed before the print process. Only those windows which have an expression are printed. Curves and bars are not printed. Depending on the printer definition in the base system (the PRIn:BOD attribute), the printout may be stored on disk.

The variable list, which may be omitted, defines the variables used in the printed picture (in windows or in start programs). In addition, the variable list may assign a value to a predefined variable called FORM\_FEED, which controls the form feed during printing. The FORM\_FEED parameter can have the following values:

- |   |  |
|---|--|
| 0 | Form feed only when the page is full (the page length is determined by the PRIn:BLP attribute, where 'n' is a physical printer number) |
| 1 | Form feed before (if not already done) and after printout  |
| 2 | Form feed before printout (if not already done)  |
| 3 | Form feed after printout   |

In conjunction with #PRINT command, the default value of FORM\_FEED is 1.

This command is not available in read-only mode.

Examples:

```
#PRINT 1 REPORT (@VAR = TEMP:PAI(1..20))
;The picture named REPORT is output to printer 1. The printer starts a
```

```
new
;page both before and after the printout.

#LOOP_WITH N = 1 .. 20
    #PRINT 1 FORM_`N` (@FORM_FEED = 0)
#LOOP_END
;The pictures FORM_1 ... FORM_20 are printed without form feed.
```

## 9.2.4 Path commands

### 9.2.4.1 #PATH name [dir [, dir]\*], #PATH name + dir [, dir]\*, #PATH name - [dir [, dir]\*]

Defines a logical path.

'name'	A logical path name, up to 10 characters.
'dir'	A directory name in the SCIL file name format (starting with /) or in the operating system format, see <a href="#">Section 6.5.1</a> for valid directory names. A trailing / or \ is stripped off, so the path can be written either with or without it. Up to 255 directories are allowed in the command. If a directory does not exist, it is automatically created. However, if the word NO_CREATE is included in the directory list, the subsequent directories are not created. If they do not exist, the error FILE_DIRECTORY_DOES_NOT_EXIST is raised.

Logical path names can be used in all commands where files are called by name, for example, together with picture names. Each path name may correspond to several directories, which are given in the search order and separated by commas. When a file is requested with the path name, it is first searched for in the first directory of the path, then in the next one, etc. If a path name is used when a new file is created, the new file will be stored in the first directory of the path name.

A #PATH command without any sign defines a global path. Such a path definition is monitor specific when executed in a picture or dialog. If executed in a command procedure, it applies to all command procedures. If executed in a printed picture, it applies to all pictures printed with #PRINT. If the directory list is omitted, all directories of the path name are removed, but the path name remains. Global paths should be defined only once, for example in the start picture or dialog. System and application specific paths should be defined by using the base system attributes SYS:BPH and APL:BPH, respectively (see the System Objects manual).

A #PATH command with a + or - sign defines local and temporary paths by adding directories to, or removing them from, the path definition. Local paths are valid only in the SCIL context where they are defined and take precedence over the global paths with the same name. A + sign means that the directories are added to the beginning of the directory list of the path name. A - sign means that the directories are removed from the directory list. If the directory list is omitted, the - sign removes all locally defined directories from the path name. (Note: as a terminating - sign means that the statement is continued on the next line, and in this case should be typed as two subsequent minus signs followed by an empty line, see the example below).

When an object invokes another one using #EXEC, #EXEC\_AFTER, #PRINT or #LIST command, both the global and local path definitions are inherited as local path definitions in the activated object. If, for example, a picture program starts a command procedure, the paths of the picture will be local paths in the command procedure and, hence, take precedence over the global command procedure paths.

The latest definition of a path is valid. Hence, by defining paths locally (with + and -), you ensure that no inherited definitions will override the desired path definitions.

The following automatically defined path names are used as default paths:

LAN_	/LAN/ACTIVE/LAN_
SYSO	/LAN/ACTIVE/SYSO
PICG	/LAN/ACTIVE/PICG
SYS_	/SYS/ACTIVE/SYS_
APL_	/APL/application/APL_
FORM	/APL/application/FORM
PICT	/APL/application/PICT

These default path names are valid everywhere, though the corresponding directories may be changed globally or locally. A file called without any path name is stored or sought in the directory/directories defined by the default path name which coincide with the first four characters in the file name. If no other path name suits, PICT is used. Visual SCIL main dialogs constitute an exception to this case. This is described in the Visual User Interface Design manual.

Defined paths can be read with the path functions, see [Section 10](#).

**Example:**

```
#PATH PROCESS + /APL/APPL1/PICT, NO_CREATE, /APL/APPL2/PICT
;A local logical path named PROCESS is created. If the directory
;/APL/APPL1/PICT does not exist, it is created, but if the directory
;/APL/APPL2/PICT does not exist, it is not created.

!NEW_PIC PROCESS/PICTURE
;The picture named PICTURE is first sought for in the application APPL1,
;then in application APPL2.

#PATH PROCESS + /APL/APPL3/PICT
!NEW_PIC PROCESS/PICTURE
;Now the directories are sought in order APPL3, APPL1, APPL2.
#PATH PROCESS - -

;empty line
;The local path definition is removed.

#PATH OWN + D:\OWN\DATAFILES
; The directory name given in operating system format.
```

#### 9.2.4.2 #REP\_LIB library [file [, file]\*], #REP\_LIB library + [file [, file]\*], #REP\_LIB library - [file [, file]\*]

Defines a logical representation library.

'library'	A logical library name, up to 10 characters.
'file'	A representation library file name, see <a href="#">Section 6.5.1</a> for valid file names. Up to 255 representation library files can be included in the command.

Logical library names can be used everywhere, where library representations are requested. Each library name may correspond to several library files, given in the search order and separated by commas. When a library representation is requested, it is first sought for in the first library, then in the next one, etc. If a library name is used when a new library representation is created, the new representation will be stored in the first file of the logical library.

A #REP\_LIB command without any sign defines a global library name. Such a name is monitor specific, when executed in a picture or dialog. If executed in a printed picture, it applies to all pictures printed with #PRINT. If the file list is omitted, all files defined for the library name are

removed, but the library name is preserved. Global libraries should be defined only once, for example in the start picture or dialog. System and application specific representation libraries should be defined by using the base system attributes SYS:BRL and APL:BRL, respectively (see the System Objects manual).

A #REP\_LIB command with a + or - sign defines local and temporary library names by adding files to or removing files from the given library name. Local representation libraries are valid only in the SCIL context where they are defined and take precedence over the global libraries with the same name. A + sign means that files are added to the beginning of the file list of the library name. A - sign means that files are removed from the list. If the file list is omitted from the command, the minus sign removes all locally added files from the actual library name. (Note: as a terminating - sign means that the statement is continued on the next line, this case should be typed as two subsequent minus signs followed by an empty line, see the example below).

When an object invokes another one using #EXEC, #EXEC\_AFTER, #PRINT or #LIST command, both the global and local representation library definitions are inherited as local library definitions in the activated object. If, for example, a picture program starts a printout, the representation libraries of the picture will be local libraries in the format picture and, hence, take precedence over the global libraries.

The latest definition of a library name is valid. Hence, by defining library names locally (with + and -), you ensure that no inherited definitions will override the desired library name definitions.

If no library name is given, a representation is searched for in the library called DEFAULT. Unless changed with a #REP\_LIB command, the library name DEFAULT corresponds to the files:

1. /APL/application/APL\_STAND.PIR
2. /LAN/ACTIVE/LAN\_/LAN\_STAND.PIR

in this search order.

Defined library names can be read with the replib functions, see [Section 10](#).

Example:

```
#PATH PATH + /APL/APPL2/PICT
#REP_LIB PROCESS PATH/MYLIB
!WIN REP WINDOW PROCESS/MYREP
;The library representation MYREP, which is to be shown in the window
WINDOW,
;is sought from the file MYLIB in the application APPL2.
```

## 9.2.5

## File handling commands

This section lists and describes the commands used to access SYS600 keyed files. Keyed files are general purpose files that may be read and written concurrently by several SCIL programs. Each data record contains a unique key that is used to identify the record. Data records may be read and written sequentially, or directly specifying the key value. See [Section 6.5.4](#) for details.

### 9.2.5.1

### #CLOSE\_FILE n

Closes a keyed file.

'n' File number. The file number assigned to the file when opened with the #OPEN\_FILE or #CREATE\_FILE command. Integer expression, 1 ... 10.

Closes the file defined by 'n'. This command should be used when the file is no longer used. Open files are automatically closed when the SCIL context is deleted.

**Example:**

```
#CLOSE_FILE 2
;The file opened as number 2 is closed.
```

### 9.2.5.2 #CREATE\_FILE n apl file keylength

Creates and opens a new keyed file.

'n'	File number. A number that identifies the file within the SCIL context. Integer expression, 1 ... 10.
'apl'	Logical application number, integer expression 0 ... 250. 0 is the own application. The application must be local.
'file'	Text or byte string expression, the name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'keylength'	The key length. Integer expression, 1 ... 253.

The command creates a keyed file with the given key length and opens the file. When no longer used it should be closed with the #CLOSE\_FILE command, see above.

This command is not available in read-only mode.

**Example:**

```
#CREATE_FILE 3 0 "RTU5" 5
;A file called RTU5 with the key length 5 is created and opened in the current ;application.
```

### 9.2.5.3 #DELETE\_FILE apl file

Deletes a file.

'apl'	Logical application number. Integer expression, 0 ... 250. 0 = the current application. The application must be local.
'file'	Text or byte string expression, the name of the file. See <a href="#">Section 6.5.1</a> for file naming.

The command deletes the named file in the given application. This command can be used to delete any file, not just keyed files.

This command is not available in read-only mode.

**Example:**

```
#DELETE_FILE 3 "RTU52" ;The file RTU5 in application 3 is deleted.
```

### 9.2.5.4 #OPEN\_FILE n apl file keylength

Opens a keyed file.

'n'	File number. A number that identifies the file within the SCIL context. Integer expression 1 ... 10.
'apl'	Logical application number, integer expression 0 ... 250. 0 = the current application. The application must be local.
'file'	Text or byte string expression, the name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'keylength'	The name of the variable to receive the key length (integer value 1 ... 253) used in the file. If a local variable by that name exists, it is used, otherwise a global variable.

The command opens a keyed file. Up to 10 files can be open at the same time in the SCIL context.

When the file is no longer used, it should be closed with the #CLOSE\_FILE command, see above.

**Example:**

```
#OPEN_FILE 2 1 "APL_/RTU3" L
;The file RTU3 in application 1 is opened. The variable L will
contain ;the key length.
```

### 9.2.5.5 #READ n key data1 [data2]

Reads a data record from a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE commands (see above). Integer expression, 1 ... 10.
'key'	The key of the record to be read. A text expression containing 'keylength' characters.
'data1', 'data2'	The names of variables to receive the data. If local variables by these names exist, they are used, otherwise global variables.

The command reads the contents of the record defined by 'key' and puts it as a text into one or two variables, 'data1' (up to 255 characters) and 'data2' (the rest of the contents if the length of the record exceeds 255 characters).

**Example:**

```
#READ 2 RTU_KEY(X:POA1) V1 V2
;The record corresponding the process object X in the opened file number
2 is
;read. The RTU_KEY function is described in .
```

### 9.2.5.6 #READ\_KEYS n keys [key1 [key2]]

Reads the keys of a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE commands (see above). Integer expression, 1 ... 10.
'keys'	The name of the variable to receive the keys. If a local variable by that name exists, it is used, otherwise a global variable.
'key1'	The first key to be read. A text expression containing up to 'keylength' characters (see the #OPEN_FILE command above). Not obligatory.
'key2'	The last key to be read. A text expression containing up to 'keylength' characters. Not obligatory.

The command reads the keys of the file defined by 'n' and stores them in the variable 'keys'. Up to 2 000 000 keys are read, starting from 'key1' and ending with 'key2'. If 'key2' is omitted the keys are read to the end of the file, or until the vector is full (2 000 000 keys). If 'key1' and 'key2' are omitted, the keys of the entire file, up to 2 000 000, are read.

**Example:**

```
#READ_KEYS 2 V "A" "B"
;Reads the keys starting with letter "A" (supposing that key length is > 1).
```

### 9.2.5.7 #READ\_NEXT n key data1 [data2]

Reads a data record from a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE commands (see above). Integer expression, 1 ... 10 .
'key'	A reference record key. A text expression.
'data1', 'data2'	The names of variables to receive the data. If local variables by these names exist, they are used, otherwise global variables.

The command reads the contents of the record next to 'key' and places it as text in one or two variables, 'data1' (up to 255 characters) and 'data2' (the rest of the contents if the length of the record exceeds 255 characters).

**Example:**

```
#READ_NEXT 2 RTU_KEY(A:POA1) V1 V2
;The record following process object A1 is read. The RTU_KEY function is described in .
```

### 9.2.5.8 #READ\_PREV n key data1 [data2]

Reads a data record from a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE commands (see above). Integer value, 1 ... 10.
'key'	A reference record key. A text expression.
'data1', 'data2'	The names of variables to receive the data. If local variables by these names exist, they are used, otherwise global variables.

The command reads the contents of the record previous to 'key' and places it as text in one or two variables, 'data1' (up to 255 characters) and 'data2' (the rest of the contents if the length of the record exceeds 255 characters).

**Example:**

```
#READ_PREV 2 RTU_KEY(A:POA1) V1 V2
;The record previous to the process object A is read. The RTU_KEY function is described in .
```

### 9.2.5.9 #REMOVE n key

Deletes a data record from a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE commands (see above). Integer value, 1 ... 10.
'key'	The key of the record to be deleted. A text containing 'keylength' characters (see the #OPEN_FILE command above).

The command deletes the chosen record. The deletion cannot be undone.

This command is not available in read-only mode.

Example:

```
#REMOVE 2 RTU_KEY(D:POA5)
;The record configuring the process object D is deleted.
```

### 9.2.5.10 #RENAME\_FILE apl old new

Renames a file.

'apl'	Logical application number. Integer value, 0 ... 250. 0 is the current application. The application must be local.
'old'	Text or byte string expression, the name of the file to be renamed. See <a href="#">Section 6.5.1</a> for file naming.
'new'	Text or byte string expression, the new file name.

The command renames the file 'old' to 'new'. If 'new' already exists, an error is raised.

This command may be used to rename any file, not just keyed files.

This command is not available in read-only mode.

Example:

```
#RENAME_FILE 0 "W89" "W90";The file W89 in the current application is
renamed to W90.
```

### 9.2.5.11 #WRITE n data1 [data2]

Writes a data record into a keyed file.

'n'	File number. The file number assigned to the file when opened with the #OPEN_FILE or #CREATE_FILE command (see above). Integer 1 ... 10.
'data1', 'data2'	Two text values containing the data to be written.

The command writes a record to the file defined by 'n'. 'data1' contains as a text the key of the record and data to be written, up to 255 characters. The rest of the data to be written, if any, is in 'data2'. The key is included as the first characters in 'data1'.

If there is a record by the same key in the file, it is overwritten.

This command is not available in read-only mode.

Example:

```
#WRITE 2 V1 V2
;The data of the variables V1 and V2 is written in the file opened as
number 2.
```

## 9.3 Visual SCIL commands

The Visual SCIL commands creates, loads and deletes the Visual SCIL objects and handles the attributes of the Visual SCIL objects. The Visual SCIL objects are superficially described in [Section 6](#) of this manual. They are described in more detail in Visual SCIL User Interface Design and Visual SCIL Objects manuals.

### 9.3.1 Loading, creating and deleting Visual SCIL objects

#### 9.3.1.1 .CREATE object = type [(attribute = value [,attribute = value]\*)]

Creates a Visual SCIL object.

'object'	The name of the object or an object reference including the path (see <a href="#">Section 6</a> ). Giving a path means that the object is created as the last child in the object chain. The object name must be unique among the objects with the same parent.
'type'	The name of the Visual SCIL object type. See the Visual SCIL Objects manual.
'attribute'	The name of an attribute which is assigned the subsequent value. If no predefined attribute with the given name exists, the attribute is created as a user defined attribute.
'value'	The value assigned to the attribute. If the attribute is predefined, the value must be of the correct data type (see the Visual SCIL Objects manual). If the attribute is user defined, the value may be of any data type.

The .CREATE command creates the object and loads it into the dialog system. Unless an object path is given, the object will be the child object of the object containing the .CREATE command.

The predefined attributes which are not assigned values in the attribute list are given default values. After an object has been created, its attributes can be changed with the .MODIFY and .SET commands, see below.

The .CREATE command is normally used for creating objects whose appearance is dynamically defined, based on some run time information. It must also be used for creating objects of the ready-built types which are not accessed in the Dialog Editor. The .LOAD command described below is used for loading dialogs and dialog items drawn in the editor.

Generally, methods cannot be written with this command. However, if the object has action methods, these can be written with attributes, see the Visual SCIL Objects manual. Except for this type of methods, objects created with .CREATE will only have the predefined methods with predefined contents.

Examples:

```
.create CANCEL = VS_BUTTON(_TITLE = "Cancel", _NOTIFY = vector(".delete
DLG"))
;The statements above creates a button with the label text Cancel and the
;NOTIFY program .delete DLG (deletes the dialog DLG).

.create ROOT\DLG = VS_DIALOG( -
    _GEOMETRY = LIST(X=100, Y=100, W=220, H=110))

.create ROOT\DLG\GROUP = VS_CONTAINER( -
    _GEOMETRY = LIST(X=5, Y=5, W=100, H=100))

.create ROOT\DLG\GROUP\ROWS = VS_CHECK_BOX( -
    _TITLE = "Check 1", -
    _GEOMETRY = LIST(X=5, Y=5, W=80, H=20))
```

```
.create ROOT\DLG\GROUP\OLUMNS = VS_CHECK_BOX( -
    _TITLE = "Check 2", -
    _GEOMETRY = LIST(X=5, Y=30, W=80, H=20))

.create ROOT\DLG\CONTAINER = VS_CONTAINER( -
    _GEOMETRY = LIST(X=115, Y=5, W=100, H=100))

.create ROOT\DLG\CONTAINER\OK = VS_BUTTON( -
    _TITLE = "OK", -
    _GEOMETRY = LIST(X=5, Y=5, W=50, H=25))

.create ROOT\DLG\CONTAINER\CANCEL = VS_BUTTON( -
    _TITLE = "Cancel", -
    _GEOMETRY = LIST(X=5, Y=35, W=50, H=25))

.create ROOT\DLG\CONTAINER\HELP = VS_BUTTON( -
    _TITLE = "Help", -
    _GEOMETRY = LIST(X=5, Y=65, W=50, H=25))

.set ROOT\DLG._OPEN = TRUE
```

The example above creates a dialog containing two containers, each of which contains dialog items. The last statement makes the dialog visible on screen.

### 9.3.1.2 .DELETE object

Deletes a Visual SCIL object.

'object'	The object name, possibly including a path (see <a href="#">Section 6</a> ).
----------	--

Deleting a Visual SCIL object means that it is removed from the screen (if shown) and from the dialog system. When an object is deleted, all its children are deleted as well.

When an object is deleted, its Delete method, if any, is executed.

Examples:

.DELETE DIALOG1	; Deletes the object DIALOG1 and all its child objects.
.DELETE ROOT	; Deletes the entire dialog system (This is a 'suicide')

### 9.3.1.3 .LOAD object = type(, file , name [, attribute = value]\*)

Loads a Visual SCIL object.

'object'	The Visual SCIL object reference (name and possibly path) of the loaded object.
	This argument determines the name of the loaded object and its situation in the object hierarchy. If the object reference is given with an object path, the loaded object will be created as the last child in the object chain. If no path is given, only object name, the object will be the child object of the object containing the .LOAD command. The object name can be freely chosen, as long as it is unique among all objects with the same parent.
'type'	The name of the Visual SCIL object type. The Visual SCIL object types are listed and described in the Visual SCIL Objects manual.

Table continues on next page

'file'	Text. The name of the file where the object is stored, including the path. Path is not needed if the file is situated in the directory defined as the default path of the main dialog of the dialog system. The default path is defined by the attribute _DEFAULT_PATH when loading the main dialog.
'name'	Text. The name under which the object is stored in the file. The name is given in the dialog editor.
'attribute'	An attribute name.
'value'	A value assigned to the attribute.

This command is used for loading objects built in the dialog editor and stored in files. It loads the specified object as a Visual SCIL object with the given name. If a loaded dialog or dialog item contains other dialog items, they are also loaded. The contained dialog items will be known as Visual SCIL objects with the names they were given in the dialog editor. The loaded object is incorporated with all its child objects into the existing object hierarchy.

When loaded, the create methods of the objects are executed (see the Visual SCIL User Interface Design manual). If the load command contains attribute definitions, the attributes are set after the create methods.

A stored dialog, dialog item or image can be concurrently loaded several times in parallel under different object names or in different contexts.

Example:

```
.load DIALOG = VS_DIALOG("MYFILE.VSO", "MYDIALOG", _OPEN = TRUE)
```

In the example above, the dialog stored as MYDIALOG in the file MYFILE is loaded as a Visual SCIL object of type VS\_DIALOG. As the visibility attribute \_OPEN is set to TRUE, the dialog is shown immediately. The command loads the complete contents of the dialog MYDIALOG.

## 9.3.2 Handling Visual SCIL attributes and methods

### 9.3.2.1 [object].method [(argument [,argument]\*)]

Calls a method.

'object'	A Visual SCIL object reference or a picture reference (see <a href="#">Section 6</a> ). THIS object, if omitted.
'method'	A method name.
'argument'	Up to 32 arguments of any data type.

Calls the specified method in the specified object, optionally with arguments. The result value returned by the called method, if any, is ignored.

### 9.3.2.2 .MODIFY object = list

Modifies one or more attributes of a Visual SCIL object.

'object'	A Visual SCIL object reference, possibly including an object path.
'list'	A list type expression, containing the attribute names and their new values. If an attribute name does not exist as a predefined attribute, it is created as a user defined attribute. For predefined attributes, the value should be of an appropriate data type. For user-defined attributes, any value will do.

Example:

```
.create CANCEL = VS_BUTTON
.modify CANCEL = list(_TITLE = "Cancel", _NOTIFY = vector(".delete DLG"))
```

### 9.3.2.3 .SET [object].attribute[component]\* = value

Assigns a value to a user interface object attribute.

'object'	A Visual SCIL object reference or a picture reference (see <a href="#">Section 6</a> ). THIS object, if omitted.
'attribute'	An attribute name.
'component'	Component of a structured value, see <a href="#">Section 5</a> . Only components of user-defined attributes may be set, the predefined attributes must be written as a whole.
'value'	The value given to the attribute or its component. If the attribute is predefined, the value must be of an allowed data type (see the attribute descriptions in the Visual SCIL Objects manual). If the attribute is user defined, any data type is allowed.

The command assigns the given attribute or its component the given value. If the object is visible when the .SET command is issued, the assignment may cause an immediate change of the appearance of the object.

When referencing Visual SCIL attributes, an error is raised if the attribute does not exist. When referencing a window attribute, the attribute is defined if it does not exist.

User defined attributes will always have the data type of the value last assigned to the attribute.

Examples:

Moving a button:

```
.SET BUTTON._GEOMETRY = (20, 20, 20, 20)
```

Defining or modifying window attributes:

```
.SET PIC_FUNC_1/WINDOW_2.RESIZED = TRUE
.SET .COUNTER(2) = 0
```

## 9.4 Picture handling commands

The picture handling commands can be used only within SYS600 pictures. They are not allowed in dialogs or in command procedures.

### 9.4.1 General picture handling commands

#### 9.4.1.1 !CLOSE

Closes the monitor.

On full graphic operator consoles, the command ends the application session by closing the kernel process of the monitor (PICO).

On a semi-graphic monitor, the command empties the screen from pictures and makes it turn black. The monitor can be refreshed by clicking the mouse, which loads the SYS600 start picture on screen. If there are pictures in the monitor alarm picture queue when the !CLOSE command is issued, the monitor screen is not emptied but instead the oldest alarm picture in

the queue is displayed on screen. If an alarm occurs while the monitor is kept closed, and the monitor in question is defined as an alarm monitor for the object, the alarm picture is immediately displayed.

#### 9.4.1.2 **!FAST\_PIC[picture], !FAST\_PIC-[picture], !FAST\_PIC[-][picture],[-[picture], ...**

Adds and removes fast picture definitions in semi-graphic monitors.

'picture'

A picture specified as:

[path/] picture name

where:

'picture name' is the picture name exclusive the extension, and

'path' is a logical path name.

If 'path' is omitted, the default path names are used. See the PATH command,  
[Section 9.2.4](#).

Default: The present picture



The command applies only to semi-graphic monitors and has no effect in full graphic monitors.

The first command model marks the picture as a fast picture. When the picture is subsequently erased from the screen, it is saved as a fast picture. This means that every dynamic part of the picture is saved in the form it had when the picture was erased. Updating is, however, terminated as in the case of ordinary pictures. When the picture is again displayed on the screen, it has the same appearance as when it was previously erased and updating starts from this state. The start program of the picture is no further executed.

The second command model, where the picture name is preceded by a minus sign, removes the fast picture definition. Henceforth, the picture is no longer stored as fast picture. Several fast picture definitions at a time can be added and removed as shown in the third command model above. Note: if 'picture' is omitted, the minus sign at the end of the line will denote that the statement is continued on the next line. To avoid this, type two subsequent minus signs and an empty line as shown in the example below.

Fast pictures are monitor specific, i.e. a picture may be a fast picture in one monitor but not in the others.

The fast pictures are continuously stored in the main memory. If there is insufficient main memory storage space in the computer, a large number of fast pictures may delay the handling of the ordinary ones.

All fast picture definitions are cancelled when the system is exited or the application state is changed. After a fast picture has been edited in the picture editor it functions in the normal way until it is stored again.

**Example:**

```
!FAST_PIC MYPIC
;Next time the picture MYPIC is exited it is stored as a fast picture.
!FAST_PIC - -
;empty line
;The fast picture definition of the current picture is cancelled.
Hereafter
;it functions as an ordinary picture.
```

### 9.4.1.3 !INT\_PIC

Displays an alarm picture.

Incoming alarms are added to a monitor (application session or window) specific alarm picture queue. This command displays the oldest alarm picture in the queue. At the same time the name of the picture is removed from the alarm picture queue. If the alarm picture queue is empty, the command has no effect.

In other respects, the command works like !NEW\_PIC.

### 9.4.1.4 !LAST\_PIC

Displays the previous picture.

This command causes the picture handling process to go one step backwards in the queue built up by !NEW\_PIC, see [Figure 17](#), and display that picture. If this picture is no longer available (for example has been deleted or is locked by Picture Editor), the previous picture in the queue is displayed. Finally, if there are no available pictures in the queue, an error is raised and no picture change occurs.

In other respects, the command works like !NEW\_PIC.

### 9.4.1.5 !NEW\_PIC picture

Displays a picture.

'picture'	The picture to be displayed, specified as [path/] picture name where: 'picture name' is the picture name exclusive the extension, and 'path' is a logical path name. If 'path' is omitted, the default path names are used. See the PATH command, <a href="#">Section 9.2.4</a> .
-----------	---

The picture with the given name, if it exists, is displayed on screen. First the background becomes visible. After that, possible draw and start programs are executed (unless the picture is stored as fast pictures, see [Section](#)). If the named picture does not exist, an error message is produced.

When a picture is displayed, its picture name is automatically placed as the last item in a monitor specific queue of shown pictures, which is maintained by the picture handling unit. If the picture name already is in the queue, all subsequent picture names are removed from the queue, and the picture name in its former position is thereafter the last item in the queue, see [Figure 17](#).

If 'path' is omitted, the picture is searched for in the directories defined by the default path name valid for the actual picture name, see the #PATH command, [Section 9.2.4](#).

An error status is produced if the picture does not exist.

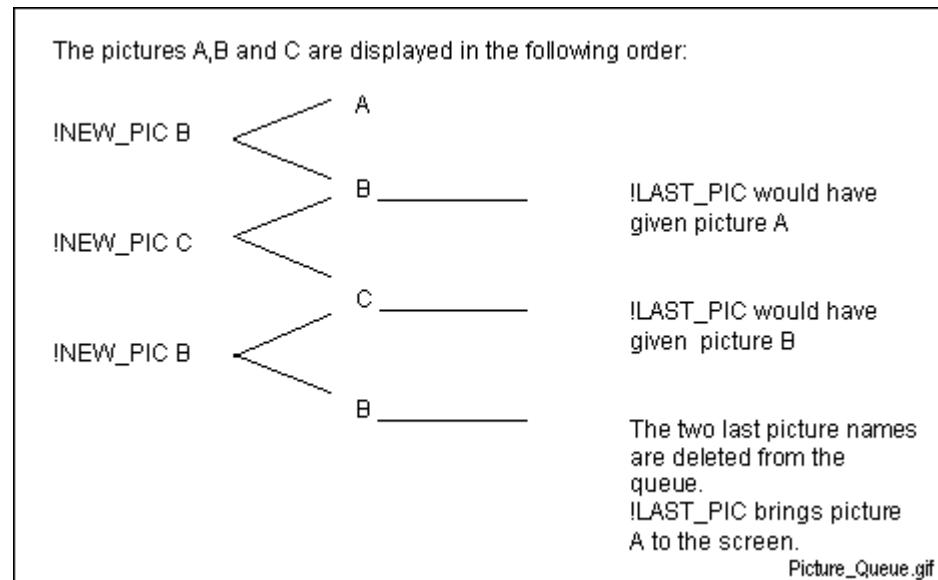
Examples:

```
!NEW_PIC MYPIC
;The picture named MYPIC is shown on screen. The picture is searched for
in
;the directory/directories defined by the path name PICT.
```

```

!NEW_PIC  OWNPIC/MYPIC
;The picture MYPIC is searched for in the directory/directories defined
by the
;path name OWNPICT.

```



*Figure 17: The picture queue. Each monitor or picture container has its own picture queue.*

#### 9.4.1.6 **!RECALL\_PIC**

Recalls a stored picture name.

The picture, the name of which is stored by !STORE\_PIC (see below), is displayed on screen. The command functions like !NEW\_PIC.

The command has no effect if !STORE\_PIC has not been used.

#### 9.4.1.7 **!RESTORE**

Stops the function key from blinking.

When a function key has been pressed, it may be marked by blinking (depending on the key definition). This command stops the blinking of all function keys on the entire screen.

#### 9.4.1.8 **!STORE\_PIC**

Stores the present picture name.

The name of the present picture is stored and it can later be retrieved with the command !RECALL\_PIC.

Only one name at a time may be stored in this way. When the command is used again, the previous picture name is removed.

#### 9.4.1.9 **!UPDATE interval**

Defines the update time interval.

'interval'	Time interval in seconds given as a positive real expression. The value of 'interval' is automatically rounded to an accuracy of 0.1 seconds.
------------	--

If there is an update program in the picture, it is executed repeatedly with this interval as long as the picture is displayed on screen, or until a new !UPDATE command is issued. After the update program has been executed to the completion, it restarts when the time interval given by this command has elapsed. Subsequent !UPDATE commands may change the time interval. The most recently given time interval is valid. If 0 is given as a time interval, the updating ceases until a new interval other than 0 is given.

When an !UPDATE command is issued, the update program is immediately executed once (provided that the interval is unlike 0). The next execution takes place at a synchronization time. After that execution restarts cyclically with the given interval. The time interval between the first and second execution may be shorter than the interval given with the !UPDATE command.

In MicroSCADA revision 8.2 no synchronization was done. If the changed policy causes troubles at an upgrade, the revision compatibility switch DO\_NOT\_SYNCHRONIZE\_PICTURE\_UPDATE may be used. See the System Objects manual for details.

The main picture, its window pictures and picture functions may have different updating intervals.

Examples:

```
!UPDATE 5
;The update program is restarted when five seconds have elapsed since the
;previous execution.
!UPDATE ;The updating interval is assigned the value of the variable A.
```

## 9.4.2 Window handling commands

Windows can be defined using the Picture Editor or the !WIN\_CREATE or !WIN\_NAME commands together with the !WIN\_POS, !WIN\_PIC, !WIN REP, and !WIN\_INPUT commands. Windows are displayed on screen with the !SHOW and !SHOW\_BACK commands and erased from screen with the !ERASE command.

When a window is displayed, it is placed on top of its parent picture (see the picture hierarchy in [Section 6](#)). Windows which are children of the same parent are arranged according to their level parameter which can be changed with the !WIN\_LEVEL command. Windows with the same parent and the same window level are stacked in the order they were first shown with the !SHOW command so that the newest window is placed on top. The updating of a window with !SHOW has no affect on the stacking order. However, when a shown window is moved by !WIN\_POS, it is raised to become the topmost window among the windows with the same level and the same parent. When the level of a displayed window is updated with the !WIN\_LEVEL command, the window is stacked as the topmost window among the windows with the same level and the same parent.

The stacking order is the same whether the child window is totally contained within its parent or not.

### 9.4.2.1 !ERASE [picture path]window

Erases a window from the screen.

'window'	The name of the window to be erased.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> . If omitted, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire picture.

The command erases the named window from the screen. If the window is of type PICTURE, a possible exit program is executed when the window is erased. If it contains windows, these windows are erased, too.

Examples:

```
!ERASE MESSAGE
;The window MESSAGE is erased.
!ERASE TREND/COLUMN
;The window COLUMN in the window (or picture function) TREND is erased.
```

#### 9.4.2.2 !SHOW [picture path]window [expression]

Shows a window.

'window'	The name of the window to be shown.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> . If omitted, the window is searched from the picture or picture function where the command was issued. If the window is not found there, the window is searched from the entire picture.
'expression'	The expression to be shown in the window. Not obligatory.

The window is displayed on screen at the location determined by the window definition. The value of the expression is calculated before the window is displayed.

If the expression is given by the !SHOW command, the expression in the window definition is disregarded. If the expression is omitted here, you may define it by the command !WIN\_INPUT. Only as the last alternative, the expression is taken from the window definition. If the window is defined as PICTURE or FIGURE, the expression is disregarded.

If the window is of PICTURE type, possible background, draw and start programs are executed when the picture is shown. The update program is executed in the interval given in the window picture, which may differ from that of the main picture.

An error status is produced if the window cannot be found.

Windows are erased with the !ERASE command.

Examples:

```
!SHOW MESSAGE "ENTER PASSWORD:"
;The text ENTER PASSWORD: is shown in the window called MESSAGE.
!SHOW TREND/COLUMN %MAX
;The variable MAX is shown in the window COLUMN of the window TREND.
```

#### 9.4.2.3 !SHOW\_BACK [picture path]window [expression]

Shows the picture background of a window.

'window'	The name of the window to be shown.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> . If omitted, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire picture.
'expression'	The expression to be shown in the window. Not obligatory.

This command shows the picture background of a window picture. The background of the picture functions included in the picture are shown as well. The background and draw programs of the picture are executed. No other programs are executed. No windows are shown and the function keys do not function.

If the window is of some other type than PICTURE, the command works like !SHOW.

An error status is produced if the window cannot be found.

The window background is erased with the !ERASE command.

Example:

```
!SHOW_BACK SCHEME
;The picture background of the window SCHEME is displayed.
```

#### 9.4.2.4 **!WIN\_BG\_COLOR [picture path]window color**

Specifies the color of the background behind the window.

'window'	The name of the window.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> . If omitted, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire picture.
'color'	The requested background color given as a named color, a vector of RGB values or a reference, see <a href="#">Section 11</a> . Default color: the color defined by the MONn:BWC attribute (see the System Objects manual).

When a window is displayed, the background color can be seen for a moment until the window is drawn on screen. Likewise, when a window is erased, the background color appears for a moment until the background behind the window is redrawn on screen. By choosing an appropriate background color for the window, disturbing color switches can be avoided.

The window background used in the monitor is specified by the base system attribute MONn:BWC. By means of the !WIN\_BG\_COLOR command, the SCIL programmer can choose the background color of individual windows. The command may be given before or after the window is shown for the first time.

When a window is shown for the first time, the background color is set to the color defined by the MON:BWC attribute, if not otherwise defined by the !WIN\_BG\_COLOR command. Later changes of WC do not affect the background color.

The !WIN\_BG\_COLOR command is not applicable to semi-graphic monitors.

#### 9.4.2.5 **!WIN\_CREATE [picture path]window**

Creates a window.

'window'	The name of the window
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> . If omitted, the window is created in the picture or picture function where the command was issued.

The command creates a new window during the picture handling. Once the window has been created with this command, it can be assigned a position with the !WIN\_POS command (default position: the upper left corner of the picture), a representation with !WIN REP (library representation) or !WIN\_PIC (picture), and an expression with the !WIN\_INPUT or !SHOW command. Similar to windows created in the picture editor, windows created with !WIN\_CREATE are shown with the !SHOW command.

!WIN\_CREATE first checks whether the window already exists. If no picture reference is given, the window is searched only within the current picture. If the window already exists, and if it is an ordinary window, the command does not produce any result. If the window exists as a Motif widget, its X background is mapped as a SYS600 window background to enable SCIL graphics to be drawn on it.

The created window exists as long as the picture is displayed on screen, or stored as fast picture (semi-graphic monitors).

Use !WIN\_CREATE instead of !WIN\_NAME in cases where !WIN\_NAME might have undesired side-effects.

#### 9.4.2.6    !WIN\_INPUT [picture path]window expression

Assigns a window an expression.

'window'	The name of the window.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'expression'	The expression to be shown in the window.

The command assigns the window an expression which is stored as long as the picture is displayed on screen or stored as a fast picture. The expression is evaluated every time the window is shown with the !SHOW command.

With this command the window expression may be changed during the program execution. The last given expression is valid. Hence, if the expression given with !WIN\_INPUT is intended to be used, no expression should be given with the !SHOW command.

If 'picture' is omitted, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire screen. An error status is produced if the window cannot be found.

Example:

```
!WIN_INPUT LINE "TRY AGAIN:"
!SHOW LINE
;The text TRY AGAIN: is shown in the window LINE.
```

#### 9.4.2.7    !WIN\_LEVEL [picture path]window level

Specifies the level parameter of the window.

'window'	The name of the window.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'level'	An integer value in the range -100 ... +100. -100 = the window is placed on bottom, +100 = the window is placed on top. Default value = 0. Instead of values, the following two predefined constants can be used: ON_BOTTOM (= -100) and ON_TOP (= +100).

The command specifies the level of the window in relation to other windows on the same level in the picture hierarchy (see [Section 6](#)). See the general part of this sub-section.

#### 9.4.2.8 !WIN\_NAME [picture path]window

Creates a new window.

'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'window'	The name of the window.

The command has the same function as the !WIN\_CREATE command, except for the following:

If no picture reference is given, the !WIN\_NAME command searches throughout the whole main picture. If a window with the given name is found, no new window is created.

Example:

```
!WIN_NAME  A
!WIN_POS  A  (1,1)
!WIN_PIC  A  A_PIC
!SHOW  A ;A window A is created and placed in the upper left corner. The
picture A_PIC
;is shown in the window.
```

#### 9.4.2.9 !WIN\_PIC [picture path]window picture

Selects a picture to be shown in the window.

'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'window'	The name of the window in which the picture will be shown.
'picture'	The name of the picture to be shown in the window, specified as [path/] picture name where: 'picture name' is the picture name exclusive the extension, and 'path' is a logical path name. If 'path' is omitted, the default path names are used. See the PATH command, <a href="#">Section 9.2.4</a> .

With this command you can change the picture name during program execution. The latest issued picture name is the valid one. The name of the window picture is stored as long as the main picture is displayed on screen or stored as a fast picture. The command entails that the picture name in the window definition is ignored.

If 'picture path' is excluded, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire screen. An error status is produced if the window cannot be found.

Example:

```

!WIN PIC AREA TABLE
!SHOW AREA
;The picture TABLE is shown in the window AREA.

```

#### 9.4.2.10 !WIN\_POS [picture path]window pos

Positions a window.

'window'	The name of the window to be positioned.
'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'pos'	The new window position as a vector with two positive integer elements. The first element is the X-position (horizontal coordinate, 1 ... 80) and the second one the Y-position (vertical coordinate, 1 ... 48). If 'pos' contains more than two elements (for example, the variable generated with the !INPUT_POS command), these elements are ignored.

The upper left corner of the window 'window' is positioned to the semi-graphic character position specified by 'pos'.

If the window is currently shown, it is immediately erased from the old location and displayed at the new one. The window is placed on the top of all windows with the same level parameter and the same parent. The new location is stored as long as the main picture is displayed on screen or stored as a fast picture.

If 'picture' is excluded from the command, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire screen. An error status is produced if the window cannot be found.

Examples:

```

!WIN_POS ABC (1,1)
;The window ABC is moved to the upper left corner of the main picture.
!SHOW INFO "POINT OUT A NEW LOCATION"
!INPUT_POS POS
!WIN_POS FIG %POS
;The window FIG is moved to the position pointed out with the mouse.

```

#### 9.4.2.11 !WIN\_REP [picture path]window representation

Selects a library representation for a window.

'picture path'	A picture path according to the format described in <a href="#">Section 6</a> .
'window'	The name of the window to be assigned a library representation.
'representation'	The library representation to be used in the window specified as: [library/] representation where: 'representation' is the name of the representation, and 'library' is a logical library name. If library name is omitted, the representation is sought from the library DEFAULT. See the command #REP_LIB in <a href="#">Section 9.2.4</a> .

With this command the representation to be used in the window may be changed during the program execution. The most recently given representation is valid. The given representation name is stored as long as the main picture is displayed or stored as a fast picture. The command entails that the representation given in the window definition is ignored.

If 'picture' is excluded from the command, the window is sought from the picture or picture function where the command was issued. If the window is not found there, the window is sought from the entire screen. An error status is produced if the window cannot be found.

Examples:

```
!WIN REP LINE  PASSWORD
!SHOW LINE
;The representation PASSWORD states the form for what is to be shown in
the
>window LINE. The representation is sought from the file(s) defined by
the
;library name DEFAULT.
!WIN REP LINE  ENG/PASSWORD
!SHOW LINE
;The same as above, but the representation is sought from the file(s)
defined
;by the library name ENG.
```

## 9.4.3 Input commands

### 9.4.3.1 !CSR\_LEFT, !CSR\_RIGHT, !CSR\_BOL, !CSR\_EOL

These commands move the data entry cursor.

!CSR\_LEFT moves the data entry cursor one step to the left and !CSR\_RIGHT one step to the right. !CSR\_BOL moves the cursor to the beginning of the input field, !CSR\_EOL to the end of the input field.

These commands may be used only after the !INPUT\_VAR command, and have no effect if this command has not been issued.

### 9.4.3.2 !ENTER

Completes data entry.

After an !INPUT\_VAR command has been executed, the system waits for character inputs until it gets an !ENTER command, or the ENTER key on the keyboard is pressed. If the program containing the !ENTER command (usually a function key program) contains other statements after !ENTER, these are executed. Then the program execution continues with the statements following after !INPUT\_VAR.

The command has no effect if no !INPUT\_VAR command has been issued.

### !INPUT\_KEY keytext var

Reads function key information.

'keytext'	This argument states the kind of information to be read:
PROGRAM	The program
HEADER	The function key header
HELP	The function key help text
'var'	A variable name. The variable will contain the selected information as a text vector. If a local variable by the name exists, it is used, otherwise a global variable.

The chosen information of the clicked function key is read and placed in the variable 'var'. The variable becomes a vector, the length of which is determined by the chosen information.

After this command has been executed, no other commands are executed until a function key has been clicked.

The execution of this command may be interrupted by input timeout (attribute APL:BIT), status PICO\_INPUT\_TIMEOUT.

**Example:**

```
!INPUT_KEY HELP V
!SHOW HELP %V
;After the !INPUT_KEY command has been executed, the system waits until a
;function key is pressed. If this has a help text, the text will be shown
in
;the window HELP.
```

#### 9.4.3.3 **!INPUT\_POS var**

Reads the mouse or cursor position.

'var'	The name of a variable. If a local variable by the name exists, it is used, otherwise a global variable.
-------	--

The command reads the coordinates of the subsequent click on the mouse, or the corresponding keyboard operation, and stores the coordinates in the given variable. After this command has been executed, no other commands are executed until a position has been pointed out with the mouse.

The variable 'var' becomes a vector with four integer elements:

(x,y,x\_rel,y\_rel)

where:

'x,y'	are the coordinates for the cursor position in relation to the upper left corner of the screen which is (1,1). The value range of the x-coordinate is 1 ... 80. The value range of the y-coordinate is 1 ... 48.
-------	--

'x_rel,y_rel'	are the coordinates of the cursor position in relation to the upper left corner (= (1,1)) of the window where the !INPUT_POS command is executed. These coordinates may have negative values.
---------------	---

The execution of this command may be interrupted by input timeout (attribute APL:BIT), status PICO\_INPUT\_TIMEOUT.

**Examples:**

```
!INPUT_POS V
;The position of the following push on the mouse is read and assigned to
the
;variable V.
!WIN_POS WINDOW %V
;The window WINDOW is moved to the position determined by the variable V
(only
;the first two elements are noted, i.e., the first coordinate pairs).
```

#### 9.4.3.4 **!INPUT\_VAR [picture path]window variable max\_length**

Reads an input value from the user.

'window'	Window name of the input field.
'picture path'	A picture reference according to the format described in <a href="#">Section 6</a> .
'variable'	Name of the variable to which the input is assigned. If a local variable by the name exists, it is used, otherwise a global variable.
'max_length'	Maximum length of the input string (in fullgraphic input fields)

This command allows the user to assign a value to a variable during operation. The window name states in which window the value is echoed. The statement is executed only if the window is defined and is of FIELD type. The window definition determines the data type of the variable (integer, real or text) as well as the number of decimals (if real data).

After this command, the system waits for an input in the form of characters, from the keyboard or a function key program, until it gets an !ENTER command or the Enter/Line Feed key of the keyboard is pressed. No commands are handled other than those affecting the character input.

Lower case letters are converted to upper case letters. The Nordic letters Å, Ä, Ö, and the German Ü are converted to the corresponding SYS600 semi-graphic characters, i.e., the ASCII codes [, \, and ].

This command is not allowed in update programs.

The execution of this command may be interrupted by input timeout (attribute APL:BIT), status PICO\_INPUT\_TIMEOUT.

Example:

```
! INPUT_VAR WIND V
;The value of the variable V is read from the window WIND.
```

#### 9.4.3.5 **!RUBOUT, !RUBOUT\_CUR, !RUBOUT\_BOL, !RUBOUT\_EOL**

Delete input data.

!RUBOUT deletes the character to the left of the input cursor.

!RUBOUT\_CUR deletes the character in the current position of the cursor.

!RUBOUT\_BOL deletes the beginning of the line until, but not including, the cursor.

!RUBOUT\_EOL deletes the end of line starting from the cursor position.

The commands may be used only after the !INPUT\_VAR command, and they have no effect if this command has not been issued.

#### **!TOGGLE\_MOD**

Insert/typeover.

Shifts from type-over (default) to insert and vice versa. The command may be used only after the !INPUT\_VAR command and has no effect if this command has not been issued.

### 9.4.4 Miscellaneous picture commands

#### 9.4.4.1 **!SEND\_PIC device number**

Copies the picture (semi-graphic) to a printer.

'device'	Device name = LPT (line printer).
'number'	Logical printer number. Integer expression, 1 ... 20.

The semi-graphic portions of the picture, in its current format, is copied to a printer (hardcopy). The printer starts a new page both before and after the printout.

Depending on the printer definition in the base system (the PRIn:BOD attribute), the printout may be stored on disk.

**Example:**

```
!SEND_PIC LPT 1
```

#### 9.4.4.2 !RESET

Deletes variables in a picture.

All variables assigned before this command, except for those of the start program, are deleted. At the same time the command !RESTORE ([Section 9.4.1](#)) is executed.

**Example:**

```
@PROFIT = 350000
!SHOW WIND %PROFIT
;350000 is shown in the window WIND.
!RESET
!SHOW WIND %PROFIT
;An error message is displayed.
#ON ERROR !RESET
;The variables are deleted when an error occurs.
```



# Section 10 Functions

This section describes the predefined SCIL functions. The section is organized into the following sections:

<a href="#">Section 10.1</a>	<a href="#">General</a>
<a href="#">Section 10.2</a>	<a href="#">Generic functions</a>
<a href="#">Section 10.3</a>	<a href="#">Arithmetic functions</a>
<a href="#">Section 10.4</a>	<a href="#">Time functions</a>
<a href="#">Section 10.5</a>	<a href="#">String functions</a>
<a href="#">Section 10.6</a>	<a href="#">Bit functions</a>
<a href="#">Section 10.7</a>	<a href="#">Vector handling functions</a>
<a href="#">Section 10.8</a>	<a href="#">List handling functions</a>
<a href="#">Section 10.9</a>	<a href="#">Functions related to program execution</a>
<a href="#">Section 10.10</a>	<a href="#">Functions related to the run-time environment</a>
<a href="#">Section 10.11</a>	<a href="#">Functions related to the programming environment</a>
<a href="#">Section 10.12</a>	<a href="#">Language functions</a>
<a href="#">Section 10.13</a>	<a href="#">Error tracing functions</a>
<a href="#">Section 10.14</a>	<a href="#">Database functions</a>
<a href="#">Section 10.15</a>	<a href="#">Network Topology Functions</a>
<a href="#">Section 10.16</a>	<a href="#">File handling functions</a>
<a href="#">Section 10.17</a>	<a href="#">File management functions</a>
<a href="#">Section 10.18</a>	<a href="#">Communication functions</a>
<a href="#">Section 10.19</a>	<a href="#">CSV (Comma Separated Value) functions</a>
<a href="#">Section 10.20</a>	<a href="#">CSV functions</a>
<a href="#">Section 10.21</a>	<a href="#">DDE client functions</a>
<a href="#">Section 10.22</a>	<a href="#">DDE server functions</a>
<a href="#">Section 10.23</a>	<a href="#">ODBC functions</a>
<a href="#">Section 10.24</a>	<a href="#">OPC Name Database functions</a>
<a href="#">Section 10.25</a>	<a href="#">OPC functions</a>
<a href="#">Section 10.26</a>	<a href="#">RTU functions</a>
<a href="#">Section 10.27</a>	<a href="#">Printout functions</a>
<a href="#">Section 10.28</a>	<a href="#">User session functions</a>
	<a href="#">Miscellaneous functions</a>

Besides the SCIL functions described in this section, there are also some functions for handling full graphic elements, which are described in [Section 11](#).

## 10.1 General

### 10.1.1 Function calls

SCIL provides a number of standard functions, which return values according to a predetermined algorithm. Function calls are used as operands in expressions (see example 8-1).

A function call has the following format:

```
function(argument(s))
```

The argument list consists of one or more expressions, separated by commas. If the function does not take any arguments, only the function name is used or an empty argument list () may be given. The expected data types of the arguments, as well as the data type of the result, are fixed by the actual function. The data type of the result determines how the function call may be used in expressions (see [Section 8.](#))

Whenever a function expects a real type argument, an integer argument will do as well. It is implicitly converted to real value before the function is called. Also, when a vector argument with real type elements is expected, integer elements are accepted as well, unless otherwise noted.

Many functions operate either on a single argument value or a vector of argument values. When a vector argument is used, each element of the resulting vector has its own status. Consequently, the function call executes without an error even if one or more of the elements of the argument vector are bad.

Several functions expect a predefined text type argument. They are called text keywords in the descriptions of functions. These arguments are case-insensitive. As an example, function LOCATE takes an optional argument by keyword value "ALL". Though not mentioned in the description, argument values "all", "All" and even "aLL" are accepted as well.

In the function descriptions in this section, the data types of the arguments, as well as the data type of the resulting value, are given for each function. Arguments within square brackets [] are optional. Notation []\* means that the argument within the square brackets is optional but may appear more than one times. If not otherwise stated, all arguments can be arbitrary SCIL expressions of the specified data type.

Examples:

The following examples illustrates the use of functions:

```
!SHOW TIME TIMES(OBJ:PRT)
;The latest registration time for the object is written out as a text in
the
>window called TIME.

#IF ODD(%I) #THEN #SET S:PBO(%I) = 0
;If the variable I is an odd number, the function ODD gets the value TRUE
and
;the #SET statement is executed.

SUM = PICK(A:D(1 .. 1000),I) + PICK(B:D(1 .. 1000),I)
;Corresponding indices of the data objects A and B are summed. The
indices are
;determined by the variable I.

!SHOW PEAK HIGH(POWER:DOV(1 .. 1000))
;The peak power is shown in the window PEAK.
```

## 10.1.2 Overview

The following table lists all SCIL functions. The last column (labeled "S") states whether the function has side-effects or not. The following notation is used:

- When the cell is empty, the function has no side-effects.
- Letter S denotes that the function has side-effects. Consequently, it can not be used in read-only mode.
- Letter P denotes that the function has potential side-effects, but only with certain argument values.

*Table 4: SCIL functions and the tasks they perform*

Function	Brief Description	Page	S
ABS(arg)	The absolute value.	<a href="#">Section 10.3.1</a>	
ADD_INTERLOCKED(object, index, amount)	Modifies the UV or SV attribute of a SYS or an APL object.	<a href="#">Section 10.28.1</a>	s
AEP_PROGRAMS(apl)	Lists the running Application Extension Programs of an application.	<a href="#">Section 10.10.1</a>	
APPEND(v, data)	Appends data to a vector.	<a href="#">Section 10.7.1</a>	
APPLICATION_ALARM_COUNT(apl [, filter])	Counts the alarms and warnings of an application.	<a href="#">Section 10.14.13</a>	
APPLICATION_ALARM_LIST(apl, lists [,attributes [, order [, filter [, max_count]]]])	Lists the alarms and warnings of an application.	<a href="#">Section 10.14.14</a>	
APPLICATION_OBJECT_ATTRIBUTES(apl, type, objects, attributes)	Reads the values of specified attributes of given application objects.	<a href="#">Section 10.14.2</a>	
APPLICATION_OBJECT_COUNT(apl, type [,order [,direction [, start [,condition]]]])	Counts application objects that fulfil given conditions.	<a href="#">Section 10.14.3</a>	
APPLICATION_OBJECT_EXISTS(apl, type, name [,condition [,verbosity]])	Checks whether an application object exists.	<a href="#">Section 10.14.4</a>	
APPLICATION_OBJECT_LIST(apl, type [,order [,direction [,start [,condition [,attributes [,max]]]]]])	Lists application objects that fulfil given conditions.	<a href="#">Section 10.14.5</a>	
APPLICATION_OBJECT_SELECT(apl, type, names, condition [,verbosity])	Selects from a list of objects the ones that fulfil the given condition.	<a href="#">Section 10.14.6</a>	
ARCCOS(arg)	Arcus cosinus.	<a href="#">Section 10.3.2</a>	
ARCSIN(arg)	Arcus sinus.	<a href="#">Section 10.3.3</a>	
ARCTAN(arg)	Arcus tangens.	<a href="#">Section 10.3.4</a>	
ARGUMENT(n)	Nth argument of the program call.	<a href="#">Section 10.9.1</a>	
ARGUMENT_COUNT	The total number of arguments of the program call.	<a href="#">Section 10.9.2</a>	
ARGUMENTS	All arguments of the program call as a vector.	<a href="#">Section 10.9.3</a>	
ASCII(n)	The Unicode character corresponding to the numeric character code.	<a href="#">Section 10.5.1</a>	
ASCII_CODE(c)	The numeric Unicode code of the character argument.	<a href="#">Section 10.5.2</a>	
ATTRIBUTE_EXISTS(list, attribute)	Checks whether a list contains given attribute(s).	<a href="#">Section 10.8.1</a>	
AUDIO_ALARM(alarm_class, on_or_off)	Sets and resets the specified audio alarm(s).	<a href="#">Section 10.28.2</a>	s
BASE_SYSTEM_OBJECT_LIST(type [,condition [,attributes [,apl]]])	Lists the base system objects that fulfil the given condition.	<a href="#">Section 10.14.7</a>	

Table continues on next page

Function	Brief Description	Page	S
BCD_TO_INTEGER(bcd)	Converts BCD coded numbers to integers.	<a href="#">Section 10.5.3</a>	
BIN(b)	Represents bit strings and integers as text in binary format.	<a href="#">Section 10.5.4</a>	
BIN_SCAN(string)	Creates an integer or real value out of its binary text representation.	<a href="#">Section 10.5.5</a>	
BINARY_SEARCH(v, value)	Searches an ordered vector by its element contents.	<a href="#">Section 10.7.2</a>	
BIT(a, b)	The bit value of a given bit in a bit string or integer.	<a href="#">Section 10.6.1</a>	
BIT_AND(a1, a2)	Bitwise logical AND of the arguments.	<a href="#">Section 10.6.2</a>	
BIT_CLEAR(a [,b]*)	Sets given bits to 0.	<a href="#">Section 10.6.3</a>	
BIT_COMPL(a)	Logical bit complement of the argument.	<a href="#">Section 10.6.4</a>	
BIT_MASK([b1 [,b]*])	Bit mask with given bits set to 1.	<a href="#">Section 10.6.5</a>	
BIT_OR(a1, a2)	Bitwise logical OR of the arguments.	<a href="#">Section 10.6.6</a>	
BIT_SCAN(string)	Creates a bit string out of its text representation.	<a href="#">Section 10.5.6</a>	
BIT_SET(a [,b]*)	Sets given bits to 1.	<a href="#">Section 10.6.7</a>	
BIT_STRING(length [,b]*)	Creates a bit string by setting given bits to 1 and the other ones to 0.	<a href="#">Section 10.6.8</a>	
BIT_XOR(a1, a2)	Bitwise logical XOR (exclusive OR) of the arguments.	<a href="#">Section 10.6.9</a>	
CAPITALIZE(text)	Capitalizes a text.	<a href="#">Section 10.5.7</a>	
CASE(selector, w <sub>1</sub> , v <sub>1</sub> [, w <sub>i</sub> , v <sub>i</sub> ]* [, "OTHERWISE", v <sub>0</sub> ])	Selects one of the arguments according to the selector argument value.	<a href="#">Section 10.2.1</a>	
CHOOSE(v) CHOOSE(v <sub>1</sub> , v <sub>2</sub> , [v <sub>i</sub> ]*)	Chooses the first non-empty argument.	<a href="#">Section 10.2.2</a>	
CLASSIFY(v, n, low, high)	Classifies the elements of a vector into size classes and returns the counts of each class.	<a href="#">Section 10.7.3</a>	
CLOCK	The present SYS time with a one-second resolution.	<a href="#">Section 10.4.2</a>	
COLLECT(v, delimiter)	Collects text fields into a text.	<a href="#">Section 10.5.8</a>	
COMPILE(source)	Runs the SCIL compiler.	<a href="#">Section 10.11.1</a>	
CONSOLE_OUTPUT(text [,severity [,category]])	Writes a message into the system error log (SYS_LOG.CSV) and into the notification window.	<a href="#">Section 10.10.2</a>	S
COS(arg)	The cosine of the argument.	<a href="#">Section 10.3.5</a>	
CSV_TO_SCIL(csv, start, field_info [,option])	Converts a CSV file format record into SCIL data.	<a href="#">Section 10.19.1</a>	
CUMULATE(v)	Accumulates the elements of the argument vector.	<a href="#">Section 10.7.4</a>	
DATA_FETCH(apl, name, index1 [,step [,count]]), DATA_FETCH(apl, name, time1, time2 [,step [,shift]]), DATA_FETCH(apl, name, time1 [,step [,count [,shift]]]), DATA_FETCH(apl, name, indices)	Reads history records of a data object.	<a href="#">Section 10.14.16</a>	
DATA_MANAGER(function [,argument]*)	SCIL database management.	<a href="#">Section 10.16.1</a>	P
DATA_MANAGER("CLOSE", handle)	Closes a SCIL database when no longer used.	<a href="#">Section 10.16.5</a>	

Table continues on next page

Function	Brief Description	Page	S
DATA_MANAGER("COPY", handle, new_file [, version])	Makes a copy of an open SCIL database into another file.	<a href="#">Section 10.16.4</a>	S
DATA_MANAGER("CREATE", file)	Creates a new SCIL database and opens it for use.	<a href="#">Section 10.16.2</a>	S
DATA_MANAGER("CREATE_SECTION", handle, section)	Creates a new (empty) section in a SCIL database.	<a href="#">Section 10.16.7</a>	S
DATA_MANAGER("DELETE", handle, section, [,component]*)	Deletes a component from a SCIL database.	<a href="#">Section 10.16.11</a>	S
DATA_MANAGER("DELETE_SECTION", handle, section)	Deletes a section, both the name and the contents, from SCIL database.	<a href="#">Section 10.16.8</a>	S
DATA_MANAGER("GET", handle, section [,component]*)	Reads data from SCIL database.	<a href="#">Section 10.16.9</a>	
DATA_MANAGER("LIST_SECTIONS", handle)	Lists the sections of a SCIL database in alphabetic order.	<a href="#">Section 10.16.6</a>	
DATA_MANAGER("OPEN", file)	Opens an existing SCIL database.	<a href="#">Section 10.16.3</a>	
DATA_MANAGER("PUT", handle, section, data [,component]*)	Writes data to SCIL database.	<a href="#">Section 10.16.10</a>	S
DATA_STORE(apl, name, data, index1 [,step]), DATA_STORE(apl, name, data, indices)	Writes historical records of a data object.	<a href="#">Section 10.14.17</a>	S
DATA_TYPE(expression)	The data type of the argument	<a href="#">Section 10.2.3</a>	
DATE[(time [, "FULL"])]	The date (year, month and day) as text.	<a href="#">Section 10.4.3</a>	
DAY[(time)]	The day of month.	<a href="#">Section 10.4.4</a>	
DDE_CONNECT(service, topic)	Opens a connection to an external application.	<a href="#">Section 10.20.1</a>	S
DDE_DISCONNECT(connection_id)	Closes the DDE connection.	<a href="#">Section 10.20.2</a>	S
DDE_EXECUTE(connection_id, statement [,timeout])	Executes a statement in a remote application.	<a href="#">Section 10.20.5</a>	S
DDE_POKE(connection_id, item, value [,timeout])	Sets the value of 'item' in a remote application.	<a href="#">Section 10.20.4</a>	S
DDE_REAL(real, separator)	Creates a DDE style real number with a user defined decimal separator.	<a href="#">Section 10.21.2</a>	
DDE_REQUEST(connection_id, item [,timeout])	Requests data from a remote application.	<a href="#">Section 10.20.3</a>	S
DDE_VECTOR(vector, decimal_separator, list_separator)	Creates a DDE style list with a user defined list separator.	<a href="#">Section 10.21.1</a>	
DEC(value [,length [,decimals]])	Represents integer and real values as text.	<a href="#">Section 10.5.9</a>	
DEC_SCAN(string)	Creates an integer or a real value out of its decimal text representation.	<a href="#">Section 10.5.10</a>	
DELETE_ATTRIBUTE(list, attribute)	Deletes attribute(s) from a list.	<a href="#">Section 10.8.2</a>	
DELETE_ELEMENT(v, index [,index2])	Deletes individual elements of a vector.	<a href="#">Section 10.7.5</a>	
DELETE_PARAMETER(file, section [,key])	Deletes a parameter from a parameter file.	<a href="#">Section 10.16.12</a>	S
DIRECTORY_MANAGER("COPY", source, target)	Copies a directory and all its contents into a new directory.	<a href="#">Section 10.17.14</a>	S
DIRECTORY_MANAGER("COPY_CONTENTS ", source, target [,filter [,subdirectories [,overwrite]]])	Copies the files of a directory into another directory. Optionally, the subdirectories are recursively copied as well.	<a href="#">Section 10.17.15</a>	S
DIRECTORY_MANAGER("CREATE", directory [,recursion])	Creates a directory or a hierarchy of directories.	<a href="#">Section 10.17.10</a>	S

Table continues on next page

Function	Brief Description	Page	S
DIRECTORY_MANAGER("DELETE", directory)	Deletes one or more directories and all the directories and files contained in them.	<a href="#">Section 10.17.11</a>	S
DIRECTORY_MANAGER("DELETE_CONTENTS", directory [,filter [,subdirectories]])	Deletes files and directories contained in a given directory.	<a href="#">Section 10.17.12</a>	S
DIRECTORY_MANAGER("EXISTS", directory)	Checks the existence of one or more directories.	<a href="#">Section 10.17.13</a>	
DIRECTORY_MANAGER("GET_ATTRIBUTES", directory)	Returns attribute information from one or more directories.	<a href="#">Section 10.17.18</a>	
DIRECTORY_MANAGER("LIST", directory [,filter [,recursion] [,hidden]])	Lists the directories contained in a given directory.	<a href="#">Section 10.17.9</a>	
DIRECTORY_MANAGER("MOVE", directory, target)	Moves a directory to another directory.	<a href="#">Section 10.17.16</a>	S
DIRECTORY_MANAGER("RENAME", directory, name)	Renames a directory.	<a href="#">Section 10.17.17</a>	S
DO(program [,a]*)	Executes the SCIL program given as an argument.	<a href="#">Section 10.9.4</a>	P
DOW[(time)]	The day of week.	<a href="#">Section 10.4.5</a>	
DOY[(time)]	The day of year .	<a href="#">Section 10.4.6</a>	
DRIVE_MANAGER("EXISTS", drive)	Checks the existence of one or more drives.	<a href="#">Section 10.17.5</a>	
DRIVE_MANAGER("GET_ATTRIBUTES", tag)	Returns some information from drives.	<a href="#">Section 10.17.7</a>	
DRIVE_MANAGER("GET_DEFAULT")	Returns the default drive, i.e. the drive assumed if an absolute path does not contain the drive.	<a href="#">Section 10.17.6</a>	
DRIVE_MANAGER("LIST")	Returns the drives available in the system.	<a href="#">Section 10.17.4</a>	
DUMP(data [,line_length])	Represents data as text in SCIL expression syntax.	<a href="#">Section 10.2.4</a>	
EDIT(text, key)	Simple text editing.	<a href="#">Section 10.5.11</a>	
ELEMENT_LENGTH(vl)	The lengths of vector elements and list attributes.	<a href="#">Section 10.2.5</a>	
END_QUERY	Tells whether a process object query is completed.	<a href="#">Section 10.14.19</a>	
ENVIRONMENT(variable)	Retrieves an operating system environment variable value.	<a href="#">Section 10.10.3</a>	
EQUAL(v1, v2, [,status_handling [,case_policy]])	Compares two SCIL values for equality.	<a href="#">Section 10.2.6</a>	
ERROR_STATE	Returns the current error handling policy.	<a href="#">Section 10.9.5</a>	
EVALUATE(expression)	Evaluates an expression given as text in SCIL syntax.	<a href="#">Section 10.2.7</a>	P
EVEN(arg)	Tells whether the argument is even.	<a href="#">Section 10.3.6</a>	
EXP(arg)	The exponential function.	<a href="#">Section 10.3.7</a>	
FETCH(apl, type, name [,index])	Fetches the configuration attributes of an object.	<a href="#">Section 10.14.9</a>	
FILE_LOCK_MANAGER(function, file)	Locks and unlocks files.	<a href="#">Section 10.16.13</a>	S
FILE_MANAGER("COPY", source, target [,overwrite])	Copies the contents of a file to another file.	<a href="#">Section 10.17.23</a>	S
FILE_MANAGER("DELETE", file)	Deletes one or more files.	<a href="#">Section 10.17.21</a>	S
FILE_MANAGER("EXISTS", file)	Checks the existence of one or more files.	<a href="#">Section 10.17.22</a>	
FILE_MANAGER("GET_ATTRIBUTES", file)	Returns attribute information from one or more files.	<a href="#">Section 10.17.26</a>	

Table continues on next page

Function	Brief Description	Page	S
FILE_MANAGER("LIST", directory [,filter [,recursion] [, hidden]])	Lists the files contained in a given directory.	<a href="#">Section 10.17.20</a>	
FILE_MANAGER("MOVE", file, target)	Moves a file to another directory.	<a href="#">Section 10.17.24</a>	S
FILE_MANAGER("RENAME", file, name)	Renames a file.	<a href="#">Section 10.17.25</a>	S
FIND_ELEMENT(v, value [,start_index [,case_policy]])	Searches a vector by its element contents.	<a href="#">Section 10.7.6</a>	
FM_APPLICATION_DIRECTORY[(path)]	Creates a directory tag out of an application relative directory path.	<a href="#">Section 10.17.28</a>	
FM_APPLICATION_FILE(path)	Creates a file tag out of an application relative file path.	<a href="#">Section 10.17.29</a>	
FM_COMBINE(tag1 [,tagi]*, tagn)	Combines two or more drive, directory or file tags to create a new directory or file tag.	<a href="#">Section 10.17.30</a>	
FM_COMBINE_NAME(name, extension)	Combines a proper file name and an extension to a file name.	<a href="#">Section 10.17.31</a>	
FM_DIRECTORY(path [,check])	Creates a directory tag out of a directory path or checks a directory path.	<a href="#">Section 10.17.32</a>	
FM_DRIVE(name [,check])	Creates a drive tag out of a drive name or checks a drive name.	<a href="#">Section 10.17.33</a>	
FM_EXTRACT(tag, component)	Extracts a component from one or more directory or file tags.	<a href="#">Section 10.17.34</a>	
FM_FILE(path [,check])	Creates a file tag out of a file path or checks a file path.	<a href="#">Section 10.17.35</a>	
FM_REPRESENT(tag [,option]*)	Converts one or more drive, directory or file tags into an OS dependent text representation.	<a href="#">Section 10.17.36</a>	
FM_SCIL_DIRECTORY(name [,check])	Creates a directory tag out of a SCIL directory name or checks a SCIL directory name.	<a href="#">Section 10.17.37</a>	
FM_SCIL_FILE(name [,option] [,option])	Creates a file tag out of a SCIL file name or checks a SCIL file name.	<a href="#">Section 10.17.38</a>	
FM_SCIL_REPRESENT(tag [,case])	Converts one or more directory or file tags into a SCIL name text representation.	<a href="#">Section 10.17.39</a>	
FM_SPLIT_NAME(file)	Extracts the proper name and the extension from one or more file names.	<a href="#">Section 10.17.40</a>	
GET_STATUS(data)	Returns the status code(s) of a value.	<a href="#">Section 10.2.8</a>	
HEX(n)	Represents an integer as text in hexadecimal format.	<a href="#">Section 10.5.12</a>	
HEX_SCAN(string)	Creates an integer or a real value out of its hexadecimal text representation.	<a href="#">Section 10.5.13</a>	
HIGH(v)	The largest (HIGH) or the smallest (LOW) element in a vector.	<a href="#">Section 10.7.7</a>	
HIGH_INDEX(v)	The index of the largest or smallest element in a vector.	<a href="#">Section 10.7.8</a>	
HIGH_PRECISION_ADD(n1 [,n]*)	Adds up two or more high precision numbers.	<a href="#">Section 10.3.8</a>	
HIGH_PRECISION_DIV(n1, n2)	Divides a high precision number by another.	<a href="#">Section 10.3.9</a>	
HIGH_PRECISION_MUL(n1, n2)	Multiplies two high precision numbers.	<a href="#">Section 10.3.10</a>	
HIGH_PRECISION_SHOW(n [,decimals])	Displays a high precision number in various formats.	<a href="#">Section 10.3.11</a>	
HIGH_PRECISION_SUB(n1, n2)	Subtracts a high precision number from another.	<a href="#">Section 10.3.12</a>	

Table continues on next page

Function	Brief Description	Page	S
HIGH_PRECISION_SUM(v)	Calculates the sum of the high precision number elements of a vector.	<a href="#">Section 10.3.13</a>	
HISTORY_DATABASE_MANAGER("CLOSE", session)	Closes a session to the history database of an application.	<a href="#">Section 10.14.23</a>	
HISTORY_DATABASE_MANAGER("GET_PARAMETERS", session)	Returns the current values of parameters.	<a href="#">Section 10.14.32</a>	
HISTORY_DATABASE_MANAGER("OPEN" [,apl])	Opens a session to the history database of an application.	<a href="#">Section 10.14.22</a>	
HISTORY_DATABASE_MANAGER("QUERY", session, count [,start])	Performs a history database query.	<a href="#">Section 10.14.33</a>	
HISTORY_DATABASE_MANAGER("READ", session, event)	Reads all the attributes of an event.	<a href="#">Section 10.14.34</a>	
HISTORY_DATABASE_MANAGER("SET_ATTRIBUTES", session, attributes)	Sets the attributes whose values are to be returned by the query.	<a href="#">Section 10.14.31</a>	
HISTORY_DATABASE_MANAGER("SET_COMMENT", session, event, comment)	Sets the EX attribute of the specified event.	<a href="#">Section 10.14.35</a>	S
HISTORY_DATABASE_MANAGER("SET_CONDITION", session, condition)	Sets the condition for requested events.	<a href="#">Section 10.14.30</a>	
HISTORY_DATABASE_MANAGER("SET_DIRECTION", session, direction)	Sets the search direction.	<a href="#">Section 10.14.28</a>	
HISTORY_DATABASE_MANAGER("SET_DIRECTORY", session, directory)	Sets the location of database files.	<a href="#">Section 10.14.25</a>	
HISTORY_DATABASE_MANAGER("SET_ORDER", session, order)	Sets the listing order.	<a href="#">Section 10.14.27</a>	
HISTORY_DATABASE_MANAGER("SET_PERIOD", session, begin [,end])	Sets the time period of the database query.	<a href="#">Section 10.14.24</a>	
HISTORY_DATABASE_MANAGER("SET_TIMEOUT", session, timeout)	Sets the maximum time a query may last.	<a href="#">Section 10.14.29</a>	
HISTORY_DATABASE_MANAGER("SET_WINDOW", session, begin, end)	Sets the time window of the query.	<a href="#">Section 10.14.26</a>	
HISTORY_DATABASE_MANAGER("WRITE", session, data)	Writes an event into the history database.	<a href="#">Section 10.14.36</a>	S
HOD[(time)]	Hours passed since the beginning of the day.	<a href="#">Section 10.4.7</a>	
HOUR[(time)]	The hour.	<a href="#">Section 10.4.8</a>	
HOY[(time)]	Hours passed since the beginning of the year.	<a href="#">Section 10.4.9</a>	
HR_CLOCK	SYS time in seconds and microseconds.	<a href="#">Section 10.4.10</a>	
IF(condition, truevalue, falsevalue)	Selects one of two arguments according to a given condition.	<a href="#">Section 10.2.9</a>	
INSERT_ELEMENT(v, pos, contents)	Inserts new elements into a vector.	<a href="#">Section 10.7.9</a>	
INTEGER_TO_BCD(int [,digits])	Represents an integer value as a BCD coded bit string	<a href="#">Section 10.5.14</a>	
INTERP(v, x)	Interpolates a value from a curve.	<a href="#">Section 10.7.10</a>	
INVERSE(v, n, low, high)	Inverts a curve.	<a href="#">Section 10.7.11</a>	
IP_PROGRAMS	Lists the running Integrated Programs in the system.	<a href="#">Section 10.10.4</a>	
JOIN(delimiter [, a]*)	Joins text fields into a text.	<a href="#">Section 10.5.15</a>	
KEYED_FILE_MANAGER(function, file [,output_file [,key_size] [,version]])	File maintenance function.	<a href="#">Section 10.16.14</a>	P

Table continues on next page

Function	Brief Description	Page	S
LENGTH(arg)	The length of the argument.	<a href="#">Section 10.2.10</a>	
LIST([attribute = expression, [attribute = expression]*])	List created out of given attribute name/value pairs.	<a href="#">Section 10.8.3</a>	
LIST_ATTR(list)	Names of attributes of a list.	<a href="#">Section 10.8.4</a>	
LN(arg)	Natural logarithm.	<a href="#">Section 10.3.14</a>	
LOCAL_TIME	The present local time.	<a href="#">Section 10.4.11</a>	
LOCAL_TIME_ADD(time, s [ms])	Adds seconds and milliseconds to given local time.	<a href="#">Section 10.4.12</a>	
LOCAL_TIME_INFORMATION[(time)]	Gives information on given local time.	<a href="#">Section 10.4.13</a>	
LOCAL_TIME_INTERVAL(from, to)	The length of the time interval between two local times.	<a href="#">Section 10.4.14</a>	
LOCAL_TO_SYS_TIME(time)	Converts local time to SYS time.	<a href="#">Section 10.4.15</a>	
LOCAL_TO_UTC_TIME(time)	Converts local time to UTC time.	<a href="#">Section 10.4.16</a>	
LOCATE(string1, string2 [, "ALL"])	Locates a text string in a text.	<a href="#">Section 10.5.16</a>	
LOGICAL_MAPPING(otype, number [, apl])	Maps physical (base system) object numbers to logical object numbers.	<a href="#">Section 10.14.40</a>	
LOW(v)	The largest (HIGH) or the smallest (LOW) element in a vector.	<a href="#">Section 10.7.7</a>	
LOWER_CASE(text)	Converts text to lower case.	<a href="#">Section 10.5.17</a>	
LOW_INDEX(v)	The index of the largest or smallest element in a vector.	<a href="#">Section 10.7.8</a>	
MAX(arg1 [,arg]*)	The largest value in the argument list.	<a href="#">Section 10.3.15</a>	
MAX_APPLICATION_NUMBER	Maximum number of application objects.	<a href="#">Section 10.11.2</a>	
MAX_BIT_STRING_LENGTH	Maximum number of bits in a bit string type value.	<a href="#">Section 10.11.3</a>	
MAX_BYTE_STRING_LENGTH	Maximum number of bytes in a byte string type value.	<a href="#">Section 10.11.4</a>	
MAX_INTEGER	Largest positive integer value.	<a href="#">Section 10.11.5</a>	
MAX_LINK_NUMBER	Maximum number of link objects.	<a href="#">Section 10.11.6</a>	
MAX_LIST_ATTRIBUTE_COUNT	Maximum number of attributes in a list.	<a href="#">Section 10.11.7</a>	
MAX_MONITOR_NUMBER	Maximum number of monitor objects.	<a href="#">Section 10.11.8</a>	
MAX_NODE_NUMBER	Maximum number of node objects.	<a href="#">Section 10.11.9</a>	
MAX_OBJECT_NAME_LENGTH	Maximum length of application and Visual SCIL object names.	<a href="#">Section 10.11.10</a>	
MAX_PICTURE_NAME_LENGTH	Maximum length of picture names.	<a href="#">Section 10.11.11</a>	
MAX_PRINTER_NUMBER	Maximum number of printer objects.	<a href="#">Section 10.11.12</a>	
MAX_PROCESS_OBJECT_INDEX	Maximum number of process objects in a process object group.	<a href="#">Section 10.11.13</a>	
MAX REPRESENTATION_NAME_LENGTH	Maximum length of representation names.	<a href="#">Section 10.11.14</a>	
MAX_STATION_NUMBER	Maximum number of station objects.	<a href="#">Section 10.11.15</a>	
MAX_STATION_TYPE_NUMBER	Maximum number of station type objects.	<a href="#">Section 10.11.16</a>	
MAX_TEXT_LENGTH	Maximum number of characters in a text type value.	<a href="#">Section 10.11.17</a>	
MAX_VECTOR_LENGTH	Maximum number of elements in a vector.	<a href="#">Section 10.11.18</a>	
MAX_WINDOW_NAME_LENGTH	Maximum length of window and picture function names.	<a href="#">Section 10.11.19</a>	

Table continues on next page

Function	Brief Description	Page	S
MEAN(v)	The mean value of the elements of a vector.	<a href="#">Section 10.7.12</a>	
MEMORY_POOL_USAGE(pool)	The amount of memory allocated from a memory pool.	<a href="#">Section 10.10.5</a>	
MEMORY_USAGE(keyword, arg)	The amount of pool memory allocated for the argument.	<a href="#">Section 10.9.6</a>	
MERGE_ATTRIBUTES(left, right)	Merges two lists into one.	<a href="#">Section 10.8.5</a>	
MIN(arg1 [,arg]*)	The smallest value in the argument list.	<a href="#">Section 10.3.16</a>	
MIN_INTEGER	Smallest negative integer value.	<a href="#">Section 10.11.20</a>	
MINUTE[(time)]	The minute.	<a href="#">Section 10.4.17</a>	
MONTH[(time)]	The number of the month.	<a href="#">Section 10.4.18</a>	
NAME_HIERARCHY(names, order, syntax [, arg4 [, arg5]])	Constructs the tree hierarchy formed by given fully qualified hierarchical names.	<a href="#">Section 10.14.38</a>	
NETWORK_TOPOLOGY_MANAGER(subfunction [, arg]* [, apl])	Manages the network topology functionality of an application.	<a href="#">Section 10.15.1</a>	P
NETWORK_TOPOLOGY_MANAGER("DELETE", model)	Deletes a model from the application.	<a href="#">Section 10.15.12</a>	S
NETWORK_TOPOLOGY_MANAGER("EXPORT", model)	Exports a model from the application	<a href="#">Section 10.15.11</a>	
NETWORK_TOPOLOGY_MANAGER("IMPORT", modeldata)	Imports a new model or updates an existing model.	<a href="#">Section 10.15.10</a>	S
NETWORK_TOPOLOGY_MANAGER("LEVELS", schema)	Lists the application specific (voltage) levels of the schema.	<a href="#">Section 10.15.4</a>	
NETWORK_TOPOLOGY_MANAGER("MODEL", model)	Returns information about a model.	<a href="#">Section 10.15.7</a>	
NETWORK_TOPOLOGY_MANAGER("MODELS")	Lists the imported network topology models of the application.	<a href="#">Section 10.15.6</a>	
NETWORK_TOPOLOGY_MANAGER("SCHEMA", schema)	Returns information about a schema.	<a href="#">Section 10.15.3</a>	
NETWORK_TOPOLOGY_MANAGER("SCHEMAS")	Lists the supported network topology schemas.	<a href="#">Section 10.15.2</a>	
NETWORK_TOPOLOGY_MANAGER("SET_LEVELS", schema, levels)	Sets the application specific (voltage) levels of the schema.	<a href="#">Section 10.15.5</a>	S
NETWORK_TOPOLOGY_MANAGER("START", model), NETWORK_TOPOLOGY_MANAGER("STOP", model)	Starts/stops calculation of a model.	<a href="#">Section 10.15.8</a>	S
NETWORK_TOPOLOGY_MANAGER("VALIDATE", modeldata)	Validates a model.	<a href="#">Section 10.15.9</a>	
NEXT(n)	Fetches the configuration attributes of an object within a search result.	<a href="#">Section 10.14.10</a>	
OBJECT_ATTRIBUTE_INFO(apl, type [,subtype [,selection]])	Returns the properties of application or system object attributes.	<a href="#">Section 10.11.21</a>	
OCT(n)	Represents an integer as text in octal format.	<a href="#">Section 10.5.18</a>	
OCT_SCAN(string)	Creates an integer or a real value out of its octal text representation.	<a href="#">Section 10.5.19</a>	
ODD(arg)	Tells whether the argument is odd.	<a href="#">Section 10.3.17</a>	
OPC_AE_ACKNOWLEDGE(apl, ln, ix, ack_id [, comment [, cookie, active_time]])	Acknowledges a condition in an OPC A&E server.	<a href="#">Section 10.24.1</a>	S

Table continues on next page

Function	Brief Description	Page	S
OPC_AE_NAMESPACE(nodenr [, clsid1 [, root]]), OPC_AE_NAMESPACE(nodename, clsid2 [, root [, user, password]])	Lists the name space of an OPC A&E server.	<a href="#">Section 10.24.2</a>	
OPC_AE_REFRESH(apl, unit)	Requests a refresh from an external OPC A&E server.	<a href="#">Section 10.24.3</a>	S
OPC_AE_SERVERS(nodenr), OPC_AE_SERVERS(nodename [, user, password])	Lists the OPC A&E servers found in a network node.	<a href="#">Section 10.24.4</a>	
OPC_AE_VALIDATE(apl, unit)	Cross-checks a process database and the name space of an OPC A&E server.	<a href="#">Section 10.24.5</a>	
OPC_DA_NAMESPACE(nodenr [, clsid1 [, root]]), OPC_DA_NAMESPACE(nodename, clsid2 [, root [, user, password]])	Lists the OPC item hierarchy of an OPC Data Access server.	<a href="#">Section 10.24.6</a>	
OPC_DA_REFRESH(apl, unit, group [, wait])	Requests a refresh of an item group from an external OPC DA server.	<a href="#">Section 10.24.7</a>	S
OPC_DA_SERVERS(nodenr), OPC_DA_SERVERS(nodename [, user, password])	Lists the OPC Data Access servers found in a network node.	<a href="#">Section 10.24.8</a>	
OPC_NAME_MANAGER(function, apl [,argument]*)	OPC Name Database maintenance.	<a href="#">Section 10.23.1</a>	P
OPC_NAME_MANAGER("DELETE", apl, name)	Deletes a name from the OPC Name Database.	<a href="#">Section 10.23.5</a>	S
OPC_NAME_MANAGER("GET", apl, name)	Reads the definition of a name from the OPC Name Database.	<a href="#">Section 10.23.4</a>	
OPC_NAME_MANAGER("LIST", apl)	Lists the OPC item names found in the OPC Name Database.	<a href="#">Section 10.23.2</a>	
OPC_NAME_MANAGER("PUT", apl, name, definition)	Creates a new name in the OPC Name Database or overwrites an existing one.	<a href="#">Section 10.23.3</a>	S
OPS_CALL(command [,nowait]), OPS_CALL(command [,option1 [,option2]])	Executes an operating system command.	<a href="#">Section 10.9.7</a>	S
OPS_NAME[[[major [,minor]]]])	Returns the name of the operating system.	<a href="#">Section 10.10.6</a>	
OPS_PROCESS(command [,directory [,option1 [,option2]]])	Starts an external program as a separate process.	<a href="#">Section 10.9.8</a>	S
PACK_STR(source, type [,length [,byte_order]])	Creates a text, a bit string or a byte string out of its elements.	<a href="#">Section 10.5.20</a>	
PACK_TIME(year, month, day, hour, minute, second)	Creates a SCIL time data value out of its components.	<a href="#">Section 10.4.19</a>	
PAD([string, ]filler, length)	Pads a text with a filler string to the given length.	<a href="#">Section 10.5.21</a>	
PARSE_FILE_NAME(name [,file])	Converts SCIL path names and file names to operating system file names.	<a href="#">Section 10.16.15</a>	
PATH(name)	The directories contained in a logical path.	<a href="#">Section 10.16.16</a>	
PATHS(level)	The logical paths defined on a specified level.	<a href="#">Section 10.16.17</a>	
PHYS_FETCH(apl, unit, address [,bit_address])	Fetches the configuration attributes of a process object	<a href="#">Section 10.14.11</a>	
PHYSICAL_MAPPING(otype, number [, apl])	Maps logical object numbers to physical (base system) object numbers.	<a href="#">Section 10.14.41</a>	
PICK(v, indices)	Picks up specified elements from a vector.	<a href="#">Section 10.7.13</a>	
PREV(n)	Fetches the configuration attributes of an object within a search result.	<a href="#">Section 10.14.10</a>	
PRINT_TRANSPARENT(data [,log])	Sends printout to a printer.	<a href="#">Section 10.26.1</a>	S

Table continues on next page

Function	Brief Description	Page	S
PRINTER_SET	Returns the target printer numbers.	<a href="#">Section 10.26.2</a>	
PROD_QUERY(n)	Returns attributes of objects selected by a process object query.	<a href="#">Section 10.14.20</a>	
RANDOM(n1, n2)	Generates a random number.	<a href="#">Section 10.3.18</a>	
READ_BYTES(file [,start [,length]])	Reads a binary file.	<a href="#">Section 10.16.18</a>	
READ_COLUMNS(file, pos, width [,start [,count]])	Reads a text file as columns.	<a href="#">Section 10.16.19</a>	
READ_PARAMETER(file, section, key [,default])	Reads a parameter from a parameter file.	<a href="#">Section 10.16.20</a>	
READ_TEXT(file [,start [,number]])	Reads a text file.	<a href="#">Section 10.16.21</a>	
REGISTRY(function, key, value_name)	Reads the registry maintained by Windows operating system.	<a href="#">Section 10.10.7</a>	P
REMOVE_DUPLICATES(v [,status_handling [,case_policy]])	Removes duplicate elements of a vector.	<a href="#">Section 10.7.14</a>	
REP_LIB(name)	The files contained in a logical representation library.	<a href="#">Section 10.16.22</a>	
REP_LIBS(level)	The logical library names defined on a specified level.	<a href="#">Section 10.16.23</a>	
REPLACE(text, string, new_string)	Replaces text strings by another string.	<a href="#">Section 10.5.22</a>	
REVERSE(v)	Reverses the order of elements of a vector.	<a href="#">Section 10.7.15</a>	
REVISION_COMPATIBILITY(issue [,enable])	Selects the compatibility issues to be used in the context.	<a href="#">Section 10.9.9</a>	
ROUND(arg [,decimals])	Rounds off a real value.	<a href="#">Section 10.3.19</a>	
RTU_ADDR(key)	Returns a list with the address of the object in a certain record.	<a href="#">Section 10.25.1</a>	
RTU_AINT(i)	Converts an integer to ASCII characters (according to the RP570 protocol).	<a href="#">Section 10.25.2</a>	
RTU_AREAL(r)	Converts a real number to four ASCII characters (float DS801).	<a href="#">Section 10.25.3</a>	
RTU_ATIME [(t [,msec])]	Converts time data (operating system time) to ASCII (RTU200 time).	<a href="#">Section 10.25.4</a>	
RTU_BIN(h)	Converts HEX-ASCII numbers given as a text to binary numbers in text form.	<a href="#">Section 10.25.5</a>	
RTU_HEXASC(b)	Converts binary numbers given as a text to hex-ascii numbers as a text.	<a href="#">Section 10.25.6</a>	
RTU_INT(a)	Converts two ASCII characters (2's complement RP570) to an integer.	<a href="#">Section 10.25.7</a>	
RTU_KEY(oa)	Returns the search key for a record in an RTU200 configuration file.	<a href="#">Section 10.25.8</a>	
RTU_MSEC(atime)	Returns the milliseconds of the 6-byte RTU time string 'atime'.	<a href="#">Section 10.25.9</a>	
RTU_OA(type, ba)	Returns the object address.	<a href="#">Section 10.25.10</a>	
RTU_REAL(a)	Converts 4 ASCII characters (float DS801) to a real number.	<a href="#">Section 10.25.11</a>	
RTU_TIME(a)	Converts ASCII (RTU200 time) to SYS600 time data.	<a href="#">Section 10.25.12</a>	
SCALE(v, scale_object [,direction])	Scales a value using a scale object.	<a href="#">Section 10.28.3</a>	
SCIL_HOST	Returns the type and number of the process that is running this SCIL code.	<a href="#">Section 10.10.8</a>	

Table continues on next page

Function	Brief Description	Page	S
SCIL_LINE_NUMBER	Tells the current line number within the SCIL program.	<a href="#">Section 10.13.1</a>	
SCIL_TO_CSV(data [,option]*)	Converts SCIL data into a CSV file format record.	<a href="#">Section 10.19.2</a>	
SECOND[(time)]	The second.	<a href="#">Section 10.4.20</a>	
SELECT(source, condition [,wildcards])	Selects the elements of a vector or a list of vectors that fulfil given condition.	<a href="#">Section 10.7.16</a>	
SEPARATE(text, delimiter)	Extracts fields of a text.	<a href="#">Section 10.5.23</a>	
SET_CLOCK(time)	Sets the SYS time.	<a href="#">Section 10.4.21</a>	
SET_EVENT_LIST_USER_NAME(name)	Sets the name that is displayed in User Name column of the event list.	<a href="#">Section 10.27.1</a>	
SET_LANGUAGE(language)	Sets the current language of the SCIL context.	<a href="#">Section 10.12.4</a>	
SET_LOCAL_TIME(time)	Sets the local time of the system.	<a href="#">Section 10.4.22</a>	S
SET_RANDOM_SEED(seed)	Sets the seed number of the random number generator.	<a href="#">Section 10.3.20</a>	
SET_STATUS(source, status)	Modifies the status code of SCIL data.	<a href="#">Section 10.2.11</a>	
SET_SYS_TIME(time)	Sets the SYS time of the system.	<a href="#">Section 10.4.23</a>	S
SET_UTC_TIME(time)	Sets the UTC time of the system.	<a href="#">Section 10.4.24</a>	S
SHADOW_FILE(file_name [,"DELETED"])	Queues a file for shadowing.	<a href="#">Section 10.16.24</a>	S
SHUFFLE(n)	Shuffles integers 1 to n into a random order.	<a href="#">Section 10.7.17</a>	
SIN(arg)	The sine of the argument.	<a href="#">Section 10.3.21</a>	
SORT(v, [start, [length]])	Sorts a vector.	<a href="#">Section 10.7.18</a>	
SPACOM(message)	Communicates with a SPACOM unit connected to a COM port.	<a href="#">Section 10.18.1</a>	S
SPREAD(v, indices, new_value)	Replaces vector elements by a new value.	<a href="#">Section 10.7.19</a>	
SQL_BEGIN_TRANSACTION(connection_id)	Marks the beginning of a transaction.	<a href="#">Section 10.22.6</a>	S
SQL_COMMIT(connection_id)	Commits a transaction the start of which was marked with SQL_BEGIN_TRANSACTION.	<a href="#">Section 10.22.7</a>	S
SQL_CONNECT(source, user, password)	Opens an ODBC connection to a data source.	<a href="#">Section 10.22.1</a>	S
SQL_DISCONNECT(connection_id)	Closes the ODBC connection defined by the argument.	<a href="#">Section 10.22.2</a>	S
SQL_EXECUTE(connection_id, SQLstring [,timeout])	Executes an SQL statement.	<a href="#">Section 10.22.3</a>	S
SQL_FETCH(statement_id)	Fetches a row of data from a result set obtained by SQL_EXECUTE.	<a href="#">Section 10.22.4</a>	S
SQL_FREE_STATEMENT(statement_id)	Frees the specified statement and stops processing associated with the statement.	<a href="#">Section 10.22.5</a>	S
SQL_ROLLBACK(connection_id)	Rolls back a transaction started with SQL_BEGIN_TRANSACTION.	<a href="#">Section 10.22.8</a>	S
SQRT(arg)	The square root of the argument.	<a href="#">Section 10.3.22</a>	
STATUS	The status code of the last error in the program.	<a href="#">Section 10.9.10</a>	
STATUS_CODE(mnemonic)	The numeric value of a mnemonic status code name.	<a href="#">Section 10.11.22</a>	
Table continues on next page			

Function	Brief Description	Page	S
STATUS_CODE_NAME(code)	The mnemonic name of a numeric status code.	<a href="#">Section 10.11.23</a>	
SUBSTR(string, start [,length])	Extracts a substring from a text, bit string or byte string value.	<a href="#">Section 10.5.24</a>	
SUM(v), SUM_POS(v), SUM_NEG(v)	The sum of all or the positive or the negative elements of a vector.	<a href="#">Section 10.7.20</a>	
SYS_TIME	The present SYS time.	<a href="#">Section 10.4.25</a>	
SYS_TIME_ADD(time, s [,ms])	Adds seconds and milliseconds to given SYS time.	<a href="#">Section 10.4.26</a>	
SYS_TIME_INTERVAL(from, to)	The length of the time interval between two SYS times.	<a href="#">Section 10.4.27</a>	
SYS_TO_LOCAL_TIME(time)	Converts SYS time to local time.	<a href="#">Section 10.4.28</a>	
SYS_TO_UTC_TIME(time)	Converts SYS time to UTC time.	<a href="#">Section 10.4.29</a>	
TEXT_READ(file [,start [,number]])	Reads a text file or a part of it.	<a href="#">Section 10.16.25</a>	
TIME[(time ["FULL"])]	Date and time as text, excluding seconds.	<a href="#">Section 10.4.30</a>	
TIME_SCAN(string [,resolution [,option1 [,option2]]])	Interprets a date/time text string.	<a href="#">Section 10.4.31</a>	
TIME_ZONE_RULES[(rule)]	Reads and sets the time zone rules of the system.	<a href="#">Section 10.4.32</a>	S
TIMEMS[(time [,msecs] ["FULL"])]	Date and time as text, including seconds and milliseconds.	<a href="#">Section 10.4.33</a>	
TIMEOUT(milliseconds)	Changes the communication time-out.	<a href="#">Section 10.18.2</a>	
TIMES[(time ["FULL"])]	Date and time as text, including seconds.	<a href="#">Section 10.4.34</a>	
TOD[(time)]	Time of day as text, excluding seconds	<a href="#">Section 10.4.35</a>	
TODMS[(time [,msecs])]	Time of day as text, including seconds and milliseconds.	<a href="#">Section 10.4.36</a>	
TODS[(time)]	Time of day as text, including seconds.	<a href="#">Section 10.4.37</a>	
TRACE_BEGIN(filename [,append] [,time_tags] [,no_cyclics])	Starts trace logging.	<a href="#">Section 10.13.2</a>	S
TRACE_END	Stops trace logging.	<a href="#">Section 10.13.3</a>	S
TRACE_PAUSE	Pauses trace logging.	<a href="#">Section 10.13.4</a>	S
TRACE_RESUME	Resumes trace logging.	<a href="#">Section 10.13.5</a>	S
TRANSLATE(text [,language])	Translates texts defined in Visual SCIL objects.	<a href="#">Section 10.12.5</a>	
TRANSLATION(id [,language])	Translates texts by using text databases.	<a href="#">Section 10.12.6</a>	
TREND(v, n)	Returns the last ('newest') elements of a vector.	<a href="#">Section 10.7.21</a>	
TRUNC(arg [,decimals])	Truncates a real value.	<a href="#">Section 10.3.23</a>	
TYPE_CAST(source, type)	Views data as if it were of different data type.	<a href="#">Section 10.2.12</a>	
UNLOCK_PICTURE(picture)	Unlocks a locked picture.	<a href="#">Section 10.28.4</a>	S
UNPACK_STR(source [,length [,byte_order]])	Splits a text, a bit string or a byte string to a vector of its elements.	<a href="#">Section 10.5.25</a>	
UPPER_CASE(text)	Converts text to upper case.	<a href="#">Section 10.5.26</a>	
USM_ADDRESS	Tells the address of the workplace computer where the user logged in.	<a href="#">Section 10.27.2</a>	
USM_AOR_DATA	Reads the AoR(Area of Responsibility) related data of the user session.	<a href="#">Section 10.27.3</a>	

Table continues on next page

Function	Brief Description	Page	S
USM_AUTHORIZATION_LEVEL(group)	Reads the authorization level of the user in a given authorization group.	<a href="#">Section 10.27.4</a>	
USM_AUTHORIZATION_LEVEL_FOR_OBJECT(group, object_name, object_index)	Reads the authorization level of the user for controlling a process object.	<a href="#">Section 10.27.5</a>	
USM_AUTHORIZATIONS	Reads the authorizations of current user.	<a href="#">Section 10.27.6</a>	
USM_CHANGE_PASSWORD(old_password, new_password)	Changes the password of the user.	<a href="#">Section 10.27.7</a>	
USM_IS_NEW_APPLICATION	Tells whether the current application is a new one.	<a href="#">Section 10.27.8</a>	
USM_LOGIN(name, password)	Logs in a user starting a new session or silently joins an existing session as a sub-session.	<a href="#">Section 10.27.9</a>	
USM_LOGOUT	Logs out the user.	<a href="#">Section 10.27.10</a>	
USM_PASSWORD_CHANGE	Tells the status of the password.	<a href="#">Section 10.27.11</a>	
USM_PASSWORD_POLICY	Reads the password policy of the application.	<a href="#">Section 10.27.12</a>	
USM_SELECT_ROLE(role)	Selects the role for the starting session.	<a href="#">Section 10.27.13</a>	
USM_SESSION_ATTRIBUTES	Reads the Monitor Pro session attributes.	<a href="#">Section 10.27.14</a>	
USM_SESSION_ID	Tells the id of current session.	<a href="#">Section 10.27.15</a>	
USM_SESSIONS	Lists the active sessions in the system.	<a href="#">Section 10.27.16</a>	
USM_USER_LANGUAGE	Tells the user's preferred language.	<a href="#">Section 10.27.17</a>	
USM_USER_NAME	Tells the name of current user.	<a href="#">Section 10.27.18</a>	
USM_USER_ROLE	Tells the name of the role that the user has selected at login.	<a href="#">Section 10.27.19</a>	
USM_USER_ROLES	Lists the roles that the user may select from after a successful login.	<a href="#">Section 10.27.20</a>	
UTC_TIME	The present UTC time.	<a href="#">Section 10.4.38</a>	
UTC_TIME_ADD(time, s [,ms])	Adds seconds and milliseconds to given UTC time.	<a href="#">Section 10.4.39</a>	
UTC_TIME_INTERVAL(from, to)	The length of the time interval between two UTC times.	<a href="#">Section 10.4.40</a>	
UTC_TO_LOCAL_TIME(time)	Converts UTC time to local time.	<a href="#">Section 10.4.41</a>	
UTC_TO_SYS_TIME(time)	Converts UTC time to SYS time.	<a href="#">Section 10.4.42</a>	
VALIDATE(as, string)	Validates a text string as a SCIL object name.	<a href="#">Section 10.11.24</a>	
VALIDATE_OBJECT_ADDRESS(apl, pt, un, oa [,subaddress] [, self])	Validates an object address.	<a href="#">Section 10.11.25</a>	
VARIABLE_NAMES	Lists the names of global variables defined in the SCIL context.	<a href="#">Section 10.9.11</a>	
VECTOR {[element1 [,element]*]}	Creates a vector out of given elements.	<a href="#">Section 10.7.22</a>	
WEEK(time)]	The number of the week within a year.	<a href="#">Section 10.4.43</a>	
WRITE_BYTES(file, data [,append])	Writes a binary file.	<a href="#">Section 10.16.26</a>	S
WRITE_COLUMNS(file, pos, width, data [,append] [,encoding])	Writes a text file as columns.	<a href="#">Section 10.16.27</a>	S
WRITE_PARAMETER(file, section, key, value [, encoding])	Writes a parameter into a parameter file.	<a href="#">Section 10.16.28</a>	S
WRITE_TEXT(file, text [,append] [,encoding])	Writes a text file.	<a href="#">Section 10.16.29</a>	S
YEAR(time)]	The year.	<a href="#">Section 10.4.44</a>	

## 10.2 Generic functions

### 10.2.1 CASE(selector, w<sub>1</sub>, v<sub>1</sub> [, w<sub>i</sub>, v<sub>i</sub>]\* [, "OTHERWISE", v<sub>o</sub>])

Selects one of the arguments according to the selector argument value.

'selector'	Any value of type integer, real, boolean, time, text or bits string.
'w <sub>1</sub> ', 'w <sub>i</sub> '	Values of a type compatible to 'selector', or a vector of such values. Up to 13 'w' arguments may be given.
'v <sub>1</sub> ', 'v <sub>i</sub> ', 'v <sub>o</sub> '	Any values.
"OTHERWISE"	An optional text keyword.
Value:	If 'selector' equals one of the 'w <sub>i</sub> ' arguments, the resulting value is the corresponding 'v <sub>i</sub> ' argument. If not, the value is 'v <sub>o</sub> ', if "OTHERWISE" argument given.

The error status SCIL\_NO\_MATCH\_IN\_CASE\_FUNCTION is returned, if no match is found and "OTHERWISE" branch is not given.

Example:

The following example gives display names to some SCIL status code values:

```
CASE(%STATUS, 0, "Good", 1, "Faulty", 2, "Obsolete", 3, "Invalid time", -  
      10, "Not sampled", "Otherwise", "Unexpected status")
```

### 10.2.2 CHOOSE(v), CHOOSE(v<sub>1</sub>, v<sub>2</sub>, [v<sub>i</sub>]\*)

Chooses the first non-empty argument.

'v'	A vector of any number of type compatible elements.
'v <sub>1</sub> ', 'v <sub>2</sub> ', 'v <sub>i</sub> '	Up to 32 type compatible arguments.
Value:	The first 'non-empty' element in the vector, or the first 'non-empty' argument in the argument list.

The meaning of 'empty' by data types:

Integer	0
Real	0.0
Boolean	FALSE
Time	1978-01-01 00:00:00
Text	""
Bit string	BIT_STRING(0)
Byte string	Empty byte string (length 0)
Vector	VECTOR()
List	LIST()

Example:

The following example evaluates to the ON attribute value of process object PO:P10, if ON is defined. If not, the result is the value of the OI attribute, if OI is defined. The result is "No id", if both ON and OI are empty.

```
CHOOSE(PO:PON10, PO:POI10, "No id")
```

### 10.2.3 DATA\_TYPE(expression)

The data type of the argument

The function returns the data type of the argument, or value "NONE" if the expression cannot be evaluated. Consequently, it can be used to check an expression, for example the existence of a variable.

'expression'	An expression of any data type.
Value:	Text. The data type of the expression:
	"INTEGER"
	"REAL"
	"BOOLEAN"
	"TEXT"
	"TIME"
	"BIT_STRING"
	"BYTE_STRING"
	"VECTOR"
	"LIST"
	"NONE" (= the expression is wrong or undefined)

Example:

```
DATA_TYPE(CLOCK) == "TIME"
#IF DATA_TYPE(DATA:DOV) == "NONE" #THEN .....
```

### 10.2.4 DUMP(data [,line\_length])

Represents data as text in SCIL expression syntax.

This function creates a text vector that represents the contents of any SCIL data in SCIL expression syntax. The status code of each data item (unless 0) is also dumped.

'data'	An expression of any data type.
'line_length'	An integer, 80 ... 65 535. Maximum line length in the resulting text vector. Default is 80.
Value:	A text vector.

This function can be used to store any SCIL data structure in a text file.

The contained control characters (codes 0 - 31) are dumped as 'ASCII(code)'.

See function EVALUATE for the reverse operation.

### 10.2.5 ELEMENT\_LENGTH(vl)

The lengths of vector elements and list attributes.

'vl'	Vector or list value.
Value:	Vector or list containing the lengths of components of 'vl'.

If the argument 'vl' is a vector, the value of the function call is an integer vector of the same length, the nth element of the resulting value containing the length of nth element of 'vl'.

If the argument 'vl' is a list, the value of the function call is a list having the same attributes as 'vl', each containing the length of the corresponding attribute of 'vl'.

**Example 1:**

```
#LOCAL LINES, LINE_LENGTHS
LINES = READ_TEXT("c:\sc\stool\misc\languages.txt") ;A text file is read
LINE_LENGTHS = ELEMENT_LENGTH(LINES) ;LINE_LENGTHS contains the length of
;each line
```

## 10.2.6 EQUAL(v1, v2, [status\_handling [,case\_policy]])

Compares two SCIL values for equality.

'v1'	First value of any type.
'v2'	Second value of any type.
'status_handling'	Text keyword value, either "CONSIDER_STATUS" or "IGNORE_STATUS", default = "IGNORE_STATUS"
'case_policy'	Text keyword value, either "CASE_SENSITIVE" or "CASE_INSENSITIVE", default = "CASE_SENSITIVE"
Value:	Boolean value TRUE if 'v1' is equal to 'v2', otherwise FALSE.

The arguments 'status\_handling' and 'case\_policy' may be given in any order.

Two values are considered equal if all the following conditions are met:

- The value types are the same.
- The values are the same.
- The status values are the same (when "CONSIDER\_STATUS") or else both are valid (<= LAST\_VALID\_STATUS) (when "IGNORE\_STATUS").

Text values are compared for equality according to the argument 'case\_policy'.

The test for equality is recursive, i.e. if the value is a vector or list, its components are tested for equality.

**Example:**

EQUAL(1, 1)	; returns TRUE
EQUAL(1, 1.0)	; returns FALSE (different types)
EQUAL(RANDOM(1, 10), RANDOM(1, 10))	; returns TRUE or FALSE
EQUAL("ABC", "abc")	; returns FALSE
EQUAL("ABC", "abc", "case_insensitive")	; returns TRUE

## 10.2.7 EVALUATE(expression)

Evaluates an expression given as text in SCIL syntax.

'expression'	A text or text vector value containing the SCIL expression to be evaluated. If a text vector is given, other lines but the last one must end with a continuation character (-).
Value:	The result of the evaluation, which may be of any data type.

See function DUMP for the reverse operation.



For any value v, EVALUATE(DUMP(v)) results to v, except for one case: real numbers may slightly lose their precision.

Example:

The following example shows how to store any data value (such as a complicated list) in a disk file and then read it back:

```
#LOCAL WRITE_STATUS = WRITE_TEXT ("DUMP.TXT", DUMP (V))
#IF WRITE_STATUS == 0 #THEN W = EVALUATE (READ_TEXT ("DUMP.TXT"))
```

## 10.2.8 GET\_STATUS(data)

Returns the status code(s) of a value.

'data'	A value of any data type.
Value:	If 'data' is of a simple data type, the function returns an integer value. If it is a vector, the function returns an integer vector containing the status codes of each element of the argument vector. If it is a list, the function returns a list with the same attribute names, each attribute containing the status code of the corresponding argument attribute.

See function SET\_STATUS for the reverse operation.

## 10.2.9 IF(condition, truevalue, falsevalue)

Selects one of two arguments according to a given condition.

'condition'	A boolean value.
'truevalue'	A value of any data type.
'falsevalue'	A value of any data type.
Value:	If 'condition' is TRUE, 'truevalue', else 'falsevalue'.

Example:

The following example evaluates to "Work" or "Weekend" according to the current day of week:

```
IF (DOW <= 5, "Work", "Weekend")
```

## 10.2.10 LENGTH (arg)

The length of the argument.

'arg'	A value of any data type.
Value:	Integer value. According to the data type of the argument, the following is returned:
Integer	1
Real	1
Boolean	1
Time	1
Text	The number of characters in the text.
Bit string	The number of bits in the string
Byte string	The number of bytes in the byte string
Vector	The number of elements in the vector.
List	The number of attributes in the list.

**Examples:**

```

LENGTH(340)      == 1
LENGTH("ABCD")   == 4
LENGTH(" ")       == 1
LENGTH("")        == 0

```

**10.2.11 SET\_STATUS(source, status)**

Modifies the status code of SCIL data.

'source'	A value of any data type.
'status'	An integer expression, 0 ... 65 535, or a vector or a list of such integers.
Value:	Same data type as 'source'.

The function merges the contents of 'source' argument and the status code(s) 'status' as follows:

- If 'source' is of a simple data type, 'status' must be an integer value.
- If 'source' is a vector, 'status' may be an integer or an integer vector. If 'status' is an integer, the status of each element of 'source' is set to 'status'. If 'status' is a vector, the status of nth element of 'source' is set to the nth element of 'status'. If the vector 'status' is shorter than the vector 'source', the 'source' is modified only up to the length of 'status'. If 'status' is longer than 'source', the extra elements are ignored.
- If 'source' is a list, 'status' may be an integer or a list value. If 'status' is an integer, the status of each attribute of 'source' is set to 'status'. If 'status' is a list, the status of 'source' attribute NN is set to the value of 'status' attribute NN. The attributes of 'status' that do not exist in 'source' are ignored. The attributes of 'source' that do not exist in 'status' are unaffected.

If a value is assigned a status code  $\geq 10$ , the value itself is lost. If a value in 'source' has a status code  $\geq 10$ , and it is assigned a status code  $< 10$ , the value is initialized by integer zero.

See function GET\_STATUS for the reverse operation.

**Examples:**

```

@V = SET_STATUS(%V, 2)
; Status code 2, OBSOLETE_STATUS, is set to the variable V.

```

```

@A = SET_STATUS(%A, GET_STATUS(%B))
; The status of the variable A is copied from the variable B.

```

## 10.2.12 TYPE\_CAST(source, type)

Views data as if it were of different data type.

'source'	The source data to be viewed. Integer, real, time, boolean, text, bit string or byte string.
'type'	Text keyword specifying the result data type: "INTEGER", "REAL", "TIME", "BOOLEAN", "TEXT", "BIT_STRING" or "BYTE_STRING" See the compatibility rules below.
Value:	A value of data type specified by 'type'.

This function interprets the memory bit pattern of a value as if it were of another data type. The following data type compatibility rules apply:

- Bit patterns of INTEGER, REAL, TIME and BOOLEAN data can be viewed as INTEGER, REAL, TIME or BOOLEAN data.
- Bit patterns of TEXT, BIT\_STRING and BYTE\_STRING data can be viewed as TEXT, BIT\_STRING or BYTE\_STRING data.
- Any other combination results to a runtime error.

This is a low-level, close-to-hardware, function, whose results may be hardware dependent. Type-casting from and to real data may result to invalid floating point numbers. Type-casting to a bit string returns the bit pattern of the source as a bit string, whose length is 8 times the length of the source (one character or byte occupies 8 bits). Type-casting from a bit string returns a text or byte string, whose length is 1/8 of the length of the bit string.

Type casting from and to TEXT values operate on 8-bit active code page encoded characters.

Examples:

TYPE\_CAST(TRUE, "INTEGER") returns the integer value 1, which is the integer representation of the boolean value TRUE.

```
@BS = PACK_STR((65, 214), "BYTE_STRING");Create a 2-element byte string
```

TYPE\_CAST(%BS, "TEXT") == "AÖ";Type casting to text results in "AÖ", if ISO Latin-1 code page is active



TYPE\_CAST function cannot be used to convert an INTEGER to a REAL number or vice versa. To convert a REAL to INTEGER, use function ROUND or TRUNC. To convert an INTEGER to REAL, add 0.0 to the integer or multiply it by 1.0.

## 10.3 Arithmetic functions

### 10.3.1 ABS(arg)

The absolute value.

'arg'	An integer or real value, or a vector of such values.
Value:	The absolute value of the argument. Same data type as the argument.

Example:

```
ABS(-3); returns the integer value 3.
```

## 10.3.2 ARCCOS(arg)

Arcus cosinus.

This function returns, as radians, the angle, whose cosine equals to the argument.

'arg'	A real value -1.0 ... +1.0, or a vector of such real values.
Value:	A real value or a vector of real values. The angle(s) in radians.

## 10.3.3 ARCSIN(arg)

Arcus sinus.

This function returns, as radians, the angle, whose sine equals to the argument.

'arg'	A real value -1.0 ... +1.0, or a vector of such real values.
Value:	A real value or a vector of real values. The angle(s) in radians.

## 10.3.4 ARCTAN(arg)

Arcus tangens.

This function returns, as radians, the angle, whose tangent equals to the argument.

'arg'	A real value, or a vector of such real values.
Value:	A real value or a vector of real values. The angle(s) in radians.

## 10.3.5 COS(arg)

The cosine of the argument.

'arg'	A real or integer value or a vector of such values. The angle(s) in radians.
Value:	A real value or a vector of real values.

## 10.3.6 EVEN(arg)

Tells whether the argument is even.

'arg'	An integer or a vector of integer values.
Value:	A boolean value or a vector of boolean values.

## 10.3.7 EXP(arg)

The exponential function.

The result is the number e, Neper's number, raised to the power of the argument.

'arg'	A real value, max. +88.0, or a vector of such values.
Value:	A real value or a vector of real values.

## 10.3.8 HIGH\_PRECISION\_ADD(n1 [,n]\*)

Adds up two or more high precision numbers.

'n1'	'DOUBLE' value, the first addend, see below.
'n'	'DOUBLE' values, up to 31 more addends.
Value:	'DOUBLE' (byte string) value, the sum of the arguments.

This and the following five functions implement the high precision arithmetics in SCIL. High precision numbers are based on 64-bit floating point numbers, while the REAL datatype of SCIL is based on 32-bit floating point numbers. High precision numbers are accurate to approx. 15 decimal digits (compared to only 7 of REAL data).

The high precision number arguments of these functions may be given in any of the following four ways (collectively called 'DOUBLE' type, for short):

1. As an integer value, for example, 1234567890.
2. As a real value, for example, 1.25.
3. As a text containing the decimal representation of the number. The representation may contain decimal digits, a sign, decimal point and leading and trailing spaces. Examples of valid text representations: "-1.234565789012", "12345678901234" and "1000000000000000000000000000000".
4. As a byte string value returned by one of the high precision functions.

The high precision functions return their high precision results as a byte string data containing the 64-bit floating point representation. This internal representation is converted to normal SCIL data types by function HIGH\_PRECISION\_SHOW.



When entering a high precision constant in the SCIL program, use a text type argument instead of a real or an integer argument, unless it is certain that the constant has an exact representation as a 32-bit floating point number or as an integer.

Example:

```
#LOCAL PI, PI_PLUS_1
PI = HIGH_PRECISION_ADD("3.141592653589793") ;Remember the double quotes!
;Otherwise you lose precision.
PI_PLUS_1 = HIGH_PRECISION_ADD(PI, 1)
```

## 10.3.9 HIGH\_PRECISION\_DIV(n1, n2)

Divides a high precision number by another.

'n1'	'DOUBLE' value, the dividend.
'n2'	'DOUBLE' value, the divisor.
Value:	'DOUBLE' (byte string) value, 'n1' / 'n2'.

See function HIGH\_PRECISION\_ADD for details of high precision arithmetics.

## 10.3.10 HIGH\_PRECISION\_MUL(n1, n2)

Multiplies two high precision numbers.

'n1'	'DOUBLE' value, the multiplier.
'n2'	'DOUBLE' value, the multiplicand.
Value:	'DOUBLE' (byte string) value, 'n1' * 'n2'.

See function HIGH\_PRECISION\_ADD for details of high precision arithmetics.

### 10.3.11 HIGH\_PRECISION\_SHOW(n [,decimals])

Displays a high precision number in various formats.

'n'	'DOUBLE' (byte string) value
'decimals'	Integer value 0 ... 253, number of decimals to display. Default value is 6.
Value:	A list with the following attributes:
REAL	Real value, 'n' truncated to a 32-bit real number.
TEXT	Text value, the decimal text representation of 'n' with 'decimals' digits after the decimal point.
INTEGRAL	Integer value, the integral part of 'n'. Real value, if the number is out of integer range.
FRACTION	Real value, the fractional part of 'n'.

See function HIGH\_PRECISION\_ADD for details of high precision arithmetics.

### 10.3.12 HIGH\_PRECISION\_SUB(n1, n2)

Subtracts a high precision number from another.

'n1'	'DOUBLE' value, the minuend.
'n2'	'DOUBLE' value, the subtrahend.
Value:	'DOUBLE' (byte string) value, 'n1' - 'n2'.

See function HIGH\_PRECISION\_ADD for details of high precision arithmetics.

### 10.3.13 HIGH\_PRECISION\_SUM(v)

Calculates the sum of the high precision number elements of a vector.

'v'	A vector of 'DOUBLE' elements.
Value:	A list with the following attributes:
SUM	'DOUBLE' (byte string) value, the calculated sum.
STATUS	Integer value, SCIL status code, 0 = OK, see below.
ERRORS	A vector of integer elements, see below.

The STATUS attribute of the result is set to SUSPICIOUS\_STATUS (= 1), if any of the elements of the argument vector has a bad status (other than 0 or 3). If the argument vector is empty, or it contains no valid elements, NOT\_SAMPLED\_STATUS is set.

If the argument vector contains elements that are not valid 'DOUBLE' values, the indices of such elements are returned in attribute ERRORS. If the length of the attribute is 0, no erroneous elements have been encountered.

See function HIGH\_PRECISION\_ADD for details of high precision arithmetics.

### 10.3.14 LN(arg)

Natural logarithm.

'arg'	A real value > 0, or a vector of such values.
Value:	A real value or a vector of real values.

### 10.3.15 MAX(arg1 [,arg]\*)

The largest value in the argument list.

'arg1' ...	Up to 32 integers, real or time values or vectors of such values.
Value:	An integer, a real or a time value or a vector of these.

The following rules apply for scalar (non-vector) arguments. First, if all the arguments are integers, the result is an integer. Second, if all the arguments are time values, the result is a time value. Third, if any of the arguments is a real value, the result is a real value. Fourth, if time arguments are mixed with numeric arguments (which actually makes no sense), the result is numeric.

Finally, if the argument list contains vectors, the result is a vector of the length of the longest argument vector. Each element is compared to the corresponding element in the other argument vectors, as well as to the scalar arguments according to the data type conversion rules above. If the lengths of argument vectors are unequal, the odd elements get SUSPICIOUS\_STATUS (see the Status Codes manual).

Notes:

1. If the argument list contains only one argument, the argument is returned as such.
2. To find the largest element of a vector, use function HIGH or HIGH\_INDEX.

See the functions MIN, HIGH and HIGH\_INDEX as well.

Examples:

```
MAX(1, 5, 3) == 5
MAX(1.0, 5, 3) == 5.0
MAX(VECTOR(1, 2, 4), 3, VECTOR(5, 0, 1)) ;returns VECTOR(5, 3, 4)
```

### 10.3.16 MIN(arg1 [,arg]\*)

The smallest value in the argument list.

'arg1' ...	Up to 32 integer, real or time values or vectors of such values.
Value:	An integer, a real or a time value or a vector of these.

The following rules apply for the scalar (non-vector) arguments. First, if all the arguments are integers, the result is an integer. Second, if all the arguments are time values, the result is a time value. Third, if any of the arguments is a real value, the result is a real value. Fourth, if time

arguments are mixed with numeric arguments (which actually makes no sense), the result is numeric.

Finally, if the argument list contains vectors, the result is a vector, which is as long as the longest argument vector. Each element is compared to the corresponding element in the other argument vectors, as well as to the scalar arguments according to the data type conversion rules mentioned above. If the lengths of argument vectors are unequal, the odd elements get SUSPICIOUS\_STATUS (see the Status Codes manual).

Notes:

1. If the argument list contains only one argument, the argument is returned as such.
2. To find the smallest element of a vector, use function LOW or LOW\_INDEX.

See the functions MAX, LOW and LOW\_INDEX as well.

Examples:

```
MIN(1, 5, 3) == 1
MIN(1.0, 5, 3) == 1.0
MIN(VECTOR(1, 2), 3, VECTOR(5, 0, 4)) ; returns VECTOR(1, 0, 3)

;The 3rd element has SUSPICIOUS_STATUS
```

### 10.3.17 ODD(arg)

Tells whether the argument is odd.

'arg'	An integer or a vector of integer values.
Value:	A boolean value or a vector of boolean values.

Example:

```
ODD(5) == TRUE
EVEN(5) == FALSE
```

### 10.3.18 RANDOM(n1, n2)

Generates a random number.

'n1'	An integer or real value.
'n2'	An integer or real value.
Value:	Integer or real value.

If 'n1' and 'n2' are integers, an integer value in the range [n1, n2] (including n1 and n2) is returned. If 'n1' and 'n2' are real values, a real value in the range [n1, n2], i.e. including n1 but excluding n2, is returned.

This function is based on the random number generator of the underlying operating system. The generator generates a fixed sequence of pseudo-random integers, where each generated number works as a seed to the next number. If you want to get exactly the same random numbers every time the SCIL program is executed. You can use function SET\_RANDOM\_SEED with a fixed argument, but if you want to have different random numbers each time, use SET\_RANDOM\_SEED with a varying argument (current time, for example).

## 10.3.19 ROUND(arg [,decimals])

Rounds off a real value.

If 'decimals' is omitted, the argument is rounded off to the nearest integer. If 'decimals' is given, the argument is rounded off to the nearest real value having 'decimals' decimal digits. See also function TRUNC.

'arg'	A real value or a vector of real values.
'decimals'	An integer $\geq 0$ , number of decimal digits
Value:	If 'decimals' omitted, an integer or a vector of integers, otherwise a real value or a vector of real values.

Examples:

```
ROUND(4.5) == 5
ROUND(-4.5) == -5
ROUND(2.7456, 2) == 2.75
```

## 10.3.20 SET\_RANDOM\_SEED(seed)

Sets the seed number of the random number generator.

'seed'	Any integer or time value.
Value:	Always integer 0.

See the function RANDOM for details.

## 10.3.21 SIN(arg)

The sine of the argument.

'arg'	A real or integer value or a vector of such values. The angle(s) in radians.
Value:	A real value or a vector of real values.

## 10.3.22 SQRT(arg)

The square root of the argument.

'arg'	A real value $\geq 0$ , or a vector of such values.
Value:	A real value $\geq 0$ or a vector of such values.

## 10.3.23 TRUNC(arg [,decimals])

Truncates a real value.

If 'decimals' is omitted, the argument is truncated to an integer by removing the fraction part. If 'decimals' is given, the argument is truncated to a real value having 'decimals' decimal digits. See also function ROUND.

'arg'	A real value or a vector of real values.
'decimals'	An integer $\geq 0$ , number of decimal digits
Value:	If 'decimals' omitted, an integer or a vector of integers, otherwise a real value or a vector of real values.

Examples:

```
TRUNC(4.5) == 4
TRUNC(-4.5) == -4
TRUNC(2.7456,2) == 2.74
```

## 10.4 Time functions

The time functions operate on time type data in various ways: They read and set system time, convert time values to a readable text representation, retrieve various information of a time type value, convert between local and UTC times and do some simple arithmetics on time values.

The time functions usually take a time type or a qualified time type argument. A time type value is used to represent a time in one-second resolution. A qualified time has a resolution of one millisecond, it also contains the daylight saving information, see below. Many of the functions may also be called without an argument, the present time is then assumed.

Both local and UTC time are supported by these functions. See function TIME\_ZONE\_RULES for details of maintaining the rules of time zone and daylight saving time transitions.

A SYS600 system may run either in local time or in UTC time. The time reference is specified by the base system attribute SYS:BTR. The term SYS time is used below to refer to the time reference of the system.

At the end of this section you find examples of the time functions.



Leap seconds are not supported by SCIL. Time 23:59:60 is never accepted nor produced by SCIL, even if it is a valid time stamp when a leap second is added according to the UTC standard. All calculations are done with 60 second minutes.

### 10.4.1 Qualified time

A qualified time represents a moment of time uniquely by adding a daylight saving time flag and status information to the one-millisecond resolution time. A qualified time is implemented as a list with predefined attributes as follows:

CL	Clock	Time type, the time in one-second resolution.
MS	Milliseconds	Integer 0 ... 999, the milliseconds
DS	Daylight saving	Boolean

Several functions that convert times from local to UTC time or otherwise calculate qualified time values return status information as the status of the CL attribute of the result. The following values may be returned:

OK\_STATUS(0). The time is existing and unique.

**FAULTY\_TIME\_STATUS** (3). The time is non-existing, either because its DS attribute is wrong or it specifies a moment that was skipped over during the transition to daylight saving time or the resulting time is out of the valid range of time type data.

**AMBIGUOUS\_TIME\_STATUS** (4). The time is ambiguous, because it specifies a moment that was duplicated during a transition to standard time and DS attribute does not tell which one is meant.

The following special cases may occur when converting UTC time to local time:

1. DS attribute of the argument is missing. This is OK, not considered as an error.
2. DS attribute is set to TRUE. The attribute is ignored and the status of resulting CL attribute is set to 3 (FAULTY\_TIME\_STATUS).
3. The resulting value of CL attribute is not in the range of TIME data type. CL is set to its minimum or maximum value and its status is set to 3 (FAULTY\_TIME\_STATUS).

The following special cases may occur when converting local time to UTC time:

1. DS attribute of the argument is missing. This is OK if the given local time is neither ambiguous nor non-existing (normally it is not). A local time may be ambiguous if it specifies a time close to the transition from daylight saving time to standard time: If the transition takes place at 4 o'clock by moving clock one hour backward, the times from 03.00.00 to 03.59.59 are ambiguous, they may represent daylight saving or standard time. If the given time is ambiguous, it is assumed that it is standard time. The status of the CL attribute of the result is set to 4 (AMBIGUOUS\_TIME\_STATUS).
2. The argument specifies a non-existing time. A local time may be non-existing if it specifies a time close to the transition from standard time to the daylight saving time. If the transition takes place at 3 o'clock by moving clock one hour forward, the times from 03.00.00 to 03.59.59 are non-existing. If the argument specifies a non-existing time, the exact UTC time of the transition is returned and the status of the CL attribute of the result is set to 3 (FAULTY\_TIME\_STATUS).
3. The DS attribute of the argument is wrong. The attribute is ignored and the status of resulting CL attribute is set to 3 (FAULTY\_TIME\_STATUS).
4. The resulting value of CL attribute is not in the range of TIME data type. CL is set to its minimum or maximum value and its status is set to 3 (FAULTY\_TIME\_STATUS).

**Example:**

```
; This function converts the given local time argument to UTC time and
; displays
; an error message if something wrong.

#ARGUMENT LOCAL_TIME_ARG
#LOCAL UTC = LOCAL_TO_UTC_TIME(LOCAL_TIME_ARG)

#CASE GET_STATUS(UTC.CL)
#WHEN STATUS_CODE("FAULTY_TIME_STATUS") -
    .SET MESSAGE._TITLE = "Skipped during transition to DST"
#WHEN STATUS_CODE("AMBIGUOUS_TIME_STATUS") -
    .SET MESSAGE._TITLE = "Please specify DST or STT"
#OTHERWISE
    .SET MESSAGE._TITLE = ""
#CASE_END
#RETURN UTC
```

## 10.4.2 CLOCK

The present SYS time with a one-second resolution.

Value:	Time value.
--------	-------------

### 10.4.3 DATE[(time , "FULL")]

The date (year, month and day) as text.

'time'	A time value or a vector of time values. Default value is the present SYS time.
"FULL"	Optional keyword argument, 4-digit year is requested.
Value:	A text or text vector.

The date is given in the format specified by attribute SYS:BTM.

If the optional argument "FULL" is given, the year is shown with 4 digits: yyyy-mm-dd, dd-mm-yyyy or mm-dd-yyyy.

See also functions TIME, TIMEMS and TIMES.

Examples:

```
DATE  
; Returns the current date, for example "03-10-22", if SYS:BTM = 0  
DATE("FULL")  
; Returns the current date, for example "22-10-2003", if SYS:BTM = 1
```

### 10.4.4 DAY[(time)]

The day of month.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1 ... 31 or a vector of such integers.

### 10.4.5 DOW[(time)]

The day of week.

The number of the day counting from Monday, which is number one.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1 ... 7 or a vector of such integers.

### 10.4.6 DOY[(time)]

The day of year .

The number of the day since the beginning of the year.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1 ... 366 or a vector of such integers.

### 10.4.7 HOD[(time)]

Hours passed since the beginning of the day.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	A real value < 24.0 or a vector of such values.

## 10.4.8 HOUR[(time)]

The hour.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 0 ... 23 or a vector of such integers.

## 10.4.9 HOY[(time)]

Hours passed since the beginning of the year.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	A real value < 8784.0 or a vector of such values.

## 10.4.10 HR\_CLOCK

SYS time in seconds and microseconds.

Value:	A list containing the following two attributes:
CL	The seconds of the SYS time as time data.
US	The microseconds of the SYS time as an integer (with an accuracy depending on the operating system, often 10 milliseconds).

This function is more or less obsolete. Use function SYS\_TIME instead.

## 10.4.11 LOCAL\_TIME

The present local time.

Value:	A list (qualified time), local time.
--------	--------------------------------------

## 10.4.12 LOCAL\_TIME\_ADD(time, s [,ms])

Adds seconds and milliseconds to given local time.

'time'	A time or list (qualified time) value, local time.
's'	An integer, seconds to add.
'ms'	An integer, milliseconds to add.
Value:	A list (qualified time) value, local time.

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

Example:

The following three calls each return the moment of time 3 seconds from now:

```
LOCAL_TIME_ADD(LOCAL_TIME, 3)
LOCAL_TIME_ADD(LOCAL_TIME, 0, 3000)
LOCAL_TIME_ADD(LOCAL_TIME, 4, -1000)
```

### 10.4.13 LOCAL\_TIME\_INFORMATION[(time)]

Gives information on given local time.

'time'	Time or list (qualified time) value, local time. Default value is the present local time.
Value:	A list with the following attributes:
UTC	Time value
DAYLIGHT_SAVING	Boolean value
TIME_ZONE	Real value
BIAS	Real value
STATUS	Integer value

The attributes contain the following information from the moment of 'time':

- UTC is the corresponding UTC time.
- DAYLIGHT\_SAVING is TRUE if daylight saving time was or is in use, otherwise FALSE.
- TIME\_ZONE is the time zone (as a real number to support fractional time zones) that the site belonged or belongs to.
- BIAS is the difference between the local time and the UTC time. When standard time is in use, BIAS is equal to TIME\_ZONE. When daylight saving time is in use, BIAS is usually TIME\_ZONE + 1.
- STATUS tells the quality of 'time':  
OK\_STATUS (0) = Existing and unique.  
FAULTY\_TIME\_STATUS (3) = Non-existing, either because its DS attribute is wrong or it specifies a moment that was skipped over during the transition to daylight saving time.  
AMBIGUOUS\_TIME\_STATUS (4) = Ambiguous, because it specifies a moment that was duplicated during the transition to standard time and DS attribute does not tell which one is meant.

### 10.4.14 LOCAL\_TIME\_INTERVAL(from, to)

The length of the time interval between two local times.

'from'	A time or list (qualified time) value, local time.
'to'	A time or list (qualified time) value, local time.
Value:	The length of the interval as a list with two attributes:
S	Integer (positive or negative), the seconds.
MS	Integer -999 ... 999, the milliseconds.

If the length of the interval is greater than MAX\_INTEGER, S is set to MAX\_INTEGER, MS is set to 999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

If the (negative) length of the interval is smaller than MIN\_INTEGER, S is set to MIN\_INTEGER, MS is set to -999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

For other possible status codes returned as the status of the S attribute of the result, see [Section 10.4.2](#).

## 10.4.15 LOCAL\_TO\_SYS\_TIME(**time**)

Converts local time to SYS time.

'time'	A time or list (qualified time) value or a vector of such values, local time(s).
Value:	A list (qualified time) or a vector of lists, SYS time(s).

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

This function either is equivalent to LOCAL\_TO\_UTC\_TIME or simply returns its argument (as qualified time value(s)), depending on the time reference of the system (SYS:BTR).

## 10.4.16 LOCAL\_TO\_UTC\_TIME(**time**)

Converts local time to UTC time.

'time'	A time or list (qualified time) value or a vector of such values, local time(s).
Value:	A list (qualified time) or a vector of lists, UTC time(s).

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

## 10.4.17 MINUTE[**(time)**]

The minute.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 0 ... 59 or a vector of such integers.

## 10.4.18 MONTH[**(time)**]

The number of the month.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1 ... 12 or a vector of such integers.

## 10.4.19 PACK\_TIME(**year, month, day, hour, minute, second**)

Creates a SCIL time data value out of its components.

'year'	Integer, 1978 ... 2113
'month'	Integer, 1 ... 12
'day'	Integer, 1 ... 31
'hour'	Integer, 0 ... 23
'minute'	Integer, 0 ... 59
'second'	Integer, 0 ... 59
Value:	Time data.

If any of the arguments is out of range, a zero (corresponding to Jan 1st 1978, 00:00 o'clock) is returned.

## 10.4.20 SECOND[(time)]

The second.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 0 ... 59 or a vector of such integers.

## 10.4.21 SET\_CLOCK(time)

Sets the SYS time.

'time'	Time data or a list with the following two attributes:
CL	The seconds of the SYS time, time value.
US	The microseconds of the SYS time, integer value.
Value:	A real value. The number of seconds that the SYS time was changed. A positive value means that the clock was set forward. A negative value indicates that it was set backward.

This function is more or less obsolete, use SET\_LOCAL\_TIME, SET\_SYS\_TIME or SET\_UTC\_TIME instead.

## 10.4.22 SET\_LOCAL\_TIME(time)

Sets the local time of the system.

'time'	A time or list (qualified time) value, local time.
Value:	The status of the operation and the resulted length of the time shift as a list with the following attributes:
STATUS	An integer. The status code, see below.
S	An integer, seconds moved.
MS	An integer -999 ... 999, milliseconds moved.

The 'time' argument is first converted to UTC time and the system clock is set accordingly. However, if the conversion results to a non-existing (FAULTY\_TIME\_STATUS) or ambiguous (AMBIGUOUS\_TIME\_STATUS) time, the system clock is not set and only the STATUS attribute is returned.

## 10.4.23 SET\_SYS\_TIME(time)

Sets the SYS time of the system.

'time'	A time or list (qualified time) value, SYS time.
Value:	The status of the operation and the resulted length of the time shift as a list with the following attributes:
STATUS	An integer. The status code, see below.
S	An integer, seconds moved.
MS	An integer -999 ... 999, milliseconds moved.

The 'time' argument is first converted to UTC time (if needed) and the system clock is set accordingly. However, if the conversion results to a non-existing (FAULTY\_TIME\_STATUS) or ambiguous (AMBIGUOUS\_TIME\_STATUS) time, the system clock is not set and only the STATUS attribute is returned.

This function is equivalent to either SET\_LOCAL\_TIME or SET\_UTC\_TIME, depending on the time reference of the system (SYS:BTR).

## 10.4.24 SET\_UTC\_TIME(time)

Sets the UTC time of the system.

'time'	A time or list (qualified time) value, UTC time.
Value:	The status of the operation and the resulted length of the time shift as a list with the following attributes:
STATUS	An integer. The status code, 0 = OK_STATUS.
S	An integer, seconds moved.
MS	An integer -999 ... 999, milliseconds moved.

## 10.4.25 SYS\_TIME

The present SYS time.

Value:	A list (qualified time), SYS time.
--------	------------------------------------

This function is equivalent to either LOCAL\_TIME or UTC\_TIME, depending on the time reference of the system (SYS:BTR).

## 10.4.26 SYS\_TIME\_ADD(time, s [,ms])

Adds seconds and milliseconds to given SYS time.

'time'	A time or list (qualified time) value, SYS time.
's'	An integer, seconds to add.
'ms'	An integer, milliseconds to add.
Value:	A list (qualified time) value, SYS time.

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

This function is equivalent to either LOCAL\_TIME\_ADD or UTC\_TIME\_ADD, depending on the time reference of the system (SYS:BTR).

Example:

The following three calls each return the moment of time 3 seconds from now:

```
SYS_TIME_ADD(SYS_TIME, 3)
SYS_TIME_ADD(SYS_TIME, 0, 3000)
SYS_TIME_ADD(SYS_TIME, 4, -1000)
```

## 10.4.27 SYS\_TIME\_INTERVAL(from, to)

The length of the time interval between two SYS times.

'from'	A time or list (qualified time) value, SYS time.
'to'	A time or list (qualified time) value, SYS time.
Value:	The length of the interval as a list with two attributes:
	S Integer (positive or negative), the seconds.
	MS Integer -999 ... 999, the milliseconds.

If the length of the interval is greater than MAX\_INTEGER, S is set to MAX\_INTEGER, MS is set to 999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

If the (negative) length of the interval is smaller than MIN\_INTEGER, S is set to MIN\_INTEGER, MS is set to -999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

For other possible status codes returned as the status of the S attribute of the result, see [Section 10.4.2](#).

This function is equivalent to either LOCAL\_TIME\_INTERVAL or UTC\_TIME\_INTERVAL, depending on the time reference of the system (SYS:BTR).

## 10.4.28 SYS\_TO\_LOCAL\_TIME(time)

Converts SYS time to local time.

'time'	A time or list (qualified time) value or a vector of such values, SYS time(s).
Value:	A list (qualified time) or a vector of lists, local time(s).

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

This function either is equivalent to UTC\_TO\_LOCAL\_TIME or simply returns its argument (as qualified time value(s)), depending on the time reference of the system (SYS:BTR).

## 10.4.29 SYS\_TO\_UTCTIME(time)

Converts SYS time to UTC time.

'time'	A time or list (qualified time) value or a vector of such values, SYS time(s).
Value:	A list (qualified time) or a vector of lists, UTC time(s).

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

This function either is equivalent to LOCAL\_TO\_UTCTIME or simply returns its argument (as qualified time value(s)), depending on the time reference of the system (SYS:BTR).

## 10.4.30 TIME[(time , "FULL")]

Date and time as text, excluding seconds.

'time'	A time value or a vector of time values. Default value is the present SYS time.
"FULL"	Optional keyword argument, 4-digit year is requested.
Value:	A text or text vector.

The date is presented in the format specified by attribute SYS:BTF.

If the optional argument "FULL" is given, the year is shown with 4 digits: yyyy-mm-dd hh:mm, dd-mm-yyyy hh:mm or mm-dd-yyyy hh:mm.

See also functions DATE, TIMEMS, TIMES and TIME\_SCAN.

Examples:

```
TIME
;Returns the current time, for example "03-10-22 12:42", if SYS:BTF = 0
TIME("FULL")
;Returns the current time, for example "22-10-2003 12:42", if SYS:BTF = 1
TIME("FULL")
;Returns the current time, for example "10-22-2003 12:42", if SYS:BTF = 2
```

## 10.4.31 TIME\_SCAN(string [,resolution [,option1 [,option2]]])

Interprets a date/time text string.

'string'	A text or text vector to be interpreted. Sample format: 2008-12-24 19:15:20.345. The date is given in the format specified by attribute SYS:BTF. The year is given with 4 or 2 digits.	
'resolution'	A text keyword, the expected resolution:	
	"DATE"	Date only
	"MINUTES"	Time without seconds
	"SECONDS"	Time with seconds
	"MILLISECONDS"	Milliseconds included (default)
'options'	One or two keywords in any order:	
	"TIME_OF_DAY"	No date expected
	"EXACT"	See note 4 below
Value:	A list with the following attributes:	
	STATUS	SCIL status code, 0 = OK
	CL	Time type (date and time)
		Integer type if "TIME_OF_DAY" (seconds since 00:00:00)
	MS	Integer, milliseconds

Notes:

1. If STATUS is non-zero, attributes CL and MS are not returned.
2. If the argument 'string' is a vector, the resulting attributes STATUS, CL and MS are vectors, too.
3. Milliseconds can be given with 1 to 3 decimals. ".100", ".10" and ".1" all denote 100 milliseconds.
4. If "EXACT" is given, the given time must be exactly in the format specified by 'resolution'. If not given, a shorter string is accepted but a longer one is flagged as an error.

For example, 2008-12-19 10:44 is valid with resolutions "MINUTES", "SECONDS" and "MILLISECONDS", if "EXACT" is not given. If "EXACT" is given, it is valid only when resolution is "MINUTES".

## 10.4.32 TIME\_ZONE\_RULES[(rule)]

Reads and sets the time zone rules of the system.

'rule'	A list value, the time zone rule to be set, see below.
Value:	A list value with two attributes:
STATUS	An integer, SCIL status code of the call.
RULES	A vector of list values, the implemented time zone rules in chronological order.

This function sets a new time zone rule and returns all the implemented rules (including the rule just set) as a vector. If the function is called without an argument, only the implemented rules are returned.

A time zone rule is represented as a list value containing the following attributes:

DATE	A time value (hours, minutes and seconds are ignored). The date after which the rule is to be applied. The default value is the present date.
TZ	An integer or real value (optional). The time zone -13 ... 13, a real value may be used to define a half or quarter hour time zone.
STT	Standard Time specification, see below (optional).
DST	Daylight Saving Time specification, see below (optional).

If any of attributes TZ, STT or DST is missing, the corresponding definition remains untouched. For example, if a given rule contains only the attribute DST, the time zone and Standard Time specification are as they were before DATE.

The Standard Time and Daylight Saving Time specification are represented as a list value containing the following attributes:

BIAS	An integer or real value. Time bias (to the time zone) while this time (STT or DST) used. The default is 0 for STT and +1 hour for DST.
MONTH	An integer 0 ... 12. The month of transition, 0 means no switch.
DAY_OF_WEEK	An integer 1 ... 7. The day of week of transition (1=Monday, 7=Sunday).
WEEK	Integer 1 ... 5. Specifies the week of the transition as follows: Suppose DAY_OF_WEEK = 7, i.e. Sunday.

Table continues on next page

Then, WEEK = 1 states that the transition takes place on the 1st Sunday of the month.

If WEEK = 5, the transition takes place on the last Sunday of the month (Value 5 has the special meaning of 'last').

HOUR	An integer 0 ... 23. The hour of transition.
MINUTE	An integer 0 ... 59. The minute of transition, default is 0.

A once only rule may be specified by omitting attributes DAY\_OF\_WEEK and WEEK, and specifying the following two attributes instead:

YEAR	Integer 1978 ... 2113. The year of the transition.
DAY	Integer 1 ... 31. The day of month of the transition.

**Example:**

Suppose the following rules have been and will be applied on a site:

1. Daylight saving time was first applied in 1983. Transition to DST at 3:00 on the last Sunday of March. Transition to STT at 4:00 on the last Sunday of September.
2. In 1997, the transition to STT was moved one month later, to the last Sunday of October.
3. In 2002 it was decided, that from 2003 on, daylight saving is no longer applied.

The following SCIL program teaches the rules to SYS600:

```
#LOCAL GENESIS = PACK_TIME(1978, 1, 1, 0, 0, 0),-
DATE1 = PACK_TIME(1983, 1, 1, 0, 0, 0),-
DATE2 = PACK_TIME(1997, 1, 1, 0, 0, 0),-
DATE3 = PACK_TIME(2003, 1, 1, 0, 0, 0)
#LOCAL RULES

;The rule obeyed before 1983

RULES = TIME_ZONE_RULES(LIST(
    DATE = GENESIS, TZ = 2,-
    DST = LIST(MONTH = 0), STT = LIST(MONTH = 0)))

;Rule 1

RULES = TIME_ZONE_RULES(LIST(
    DATE = DATE1,-
    DST = LIST(MONTH = 3, DAY_OF_WEEK = 7, WEEK = 5, HOUR = 3),-
    STT = LIST(MONTH = 9, DAY_OF_WEEK = 7, WEEK = 5, HOUR = 4)))

;Rule 2

RULES = TIME_ZONE_RULES(LIST(
    DATE = DATE2,-
    STT = LIST(MONTH = 10, DAY_OF_WEEK = 7, WEEK = 5, HOUR = 4)))

;Rule 3

RULES = TIME_ZONE_RULES(LIST(
    DATE = DATE3,-
    DST=LIST(MONTH = 0), STT=LIST(MONTH = 0)))
```

The time zone rules are stored in the text file SYS-/SYS\_TIME.PAR. After running the example above, the contents of the file are as follows (the second rule has been divided to two lines here, for clarity):

```
1978-01-01 DST=LIST(MONTH=0),STT=LIST(MONTH=0),TZ=2
1983-01-01 DST=LIST(BIAS=1,DAY_OF_WEEK=7,HOUR=3,MINUTE=0,MONTH=3,WEEK=5),-
STT=LIST(BIAS=0,DAY_OF_WEEK=7,HOUR=4,MINUTE=0,MONTH=9,WEEK=5)
```

```
1997-01-01 STT=LIST(BIAS=0, DAY_OF_WEEK=7, HOUR=4, MINUTE=0, MONTH=10, WEEK=5)
2003-01-01 DST=LIST(MONTH=0), STT=LIST(MONTH=0)
```

Whenever the SYS600 program is restarted, the current rule obeyed by the operating system is read. If it differs from the current rule from SYS\_TIME.PAR, a new rule is appended to the file. This makes an alternative way to add a new time zone rule: change the operating system time zone settings and restart SYS600. However, a rule obeyed in the past cannot be added this way and the program must be restarted.

When the program is started for the first time, it makes the sophisticated guess that the current rule has been applied since the beginning of SYS600 time counting, 1st of January, 1978.

### 10.4.33 **TIMEMS[(time [,msecs] ,,"FULL")]**

Date and time as text, including seconds and milliseconds.

'time'	A time value, a qualified time (list) value or a vector of such values. The default value is the present SYS time.
'msecs'	An optional integer or integer vector argument, milliseconds 0 ... 999.
"FULL"	An optional keyword argument, 4-digit year is requested.
Value:	A text or text vector.

The date is presented in the format specified by attribute SYS:BTF.

If the optional argument "FULL" is given, the year is shown with 4 digits: yyyy-mm-dd hh:mm:ss.mmm, dd-mm-yyyy hh:mm:ss.mmm or mm-dd-yyyy hh:mm:ss.mmm.

If the milliseconds are given explicitly as a vector, the length of the vector must be equal to the length of the 'time' vector.

See also the DATE, TIME, TIMES, TIME\_SCAN, TOD, TODMS and TODS functions.

Examples:

```
TIMEMS(ABC:PRQ1)
; Returns the registration time of the process object ABC:P1
; for example "04-03-22 12:42:15.030" (SYS:BTF = 0)
TIMEMS("FULL")
; Returns the current time, for example "22-03-2004 12:42:15.030" (SYS:BTF
= 1)
```

### 10.4.34 **TIMES[(time ,,"FULL")]**

Date and time as text, including seconds.

'time'	A time value or a vector of time values. Default value is the present SYS time.
"FULL"	Optional keyword argument, 4-digit year is requested.
Value:	A text or text vector.

The date is presented in the format specified by attribute SYS:BTF.

If the optional argument "FULL" is given, the year is shown with 4 digits: yyyy-mm-dd hh:mm:ss, dd-mm-yyyy hh:mm:ss or mm-dd-yyyy hh:mm:ss.

See also functions DATE, TIME, TIMEMS, TIME\_SCAN, TOD, TODMS and TODS.

**Examples:**

```

TIMES
;Returns the current time, for example "03-10-22 12:42:15", if SYS:BTF = 0
TIMES("FULL")
;Returns the current time, for example "22-10-2003 12:42:15", if SYS:BTF
= 1

```

## 10.4.35 **TOD[(time)]**

Time of day as text, excluding seconds

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	A text or text vector, in format hh:mm.

## 10.4.36 **TODMS[(time [,msecs])]**

Time of day as text, including seconds and milliseconds.

'time'	A time value, a qualified time (list) value or a vector of such values. The default value is the present SYS time.
'msecs'	An optional integer or integer vector argument, milliseconds 0 ... 999.
Value:	A text or text vector, in format hh:mm:ss.mmmm.

If the milliseconds are given explicitly as a vector, the length of the vector must equal to the length of the 'time' vector.

## 10.4.37 **TODS[(time)]**

Time of day as text, including seconds.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	A text or text vector, in format hh:mm:ss.

## 10.4.38 **UTC\_TIME**

The present UTC time.

Value:	A list (qualified time), UTC time.
--------	------------------------------------

## 10.4.39 **UTC\_TIME\_ADD(time, s [,ms])**

Adds seconds and milliseconds to given UTC time.

'time'	A time or list (qualified time) value, UTC time.
's'	An integer, seconds to add.
'ms'	An integer, milliseconds to add.
Value:	A list (qualified time) value, UTC time.

For the possible status codes returned as the status of the CL attribute of the result, see [Section 10.4.2](#).

Example:

The following three calls each return the moment of time 3 seconds from now:

```
UTC_TIME_ADD(UTC_TIME, 3)
UTC_TIME_ADD(UTC_TIME, 0, 3000)
UTC_TIME_ADD(UTC_TIME, 4, -1000)
```

## 10.4.40 UTC\_TIME\_INTERVAL(from, to)

The length of the time interval between two UTC times.

'from'	A time or list (qualified time) value, UTC time.
'to'	A time or list (qualified time) value, UTC time.
Value:	The length of the interval as a list with two attributes:
	S Integer (positive or negative), the seconds.
	MS Integer -999 ... 999, the milliseconds.

If the length of the interval is greater than MAX\_INTEGER, S is set to MAX\_INTEGER, MS is set to 999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

If the (negative) length of the interval is smaller than MIN\_INTEGER, S is set to MIN\_INTEGER, MS is set to -999 and the status of S is set to 3 (FAULTY\_TIME\_STATUS).

## 10.4.41 UTC\_TO\_LOCAL\_TIME(time)

Converts UTC time to local time.

'time'	A time or list (qualified time) value or a vector of such values, UTC time(s).
Value:	A list (qualified time) or a vector of lists, local time(s).

## 10.4.42 UTC\_TO\_SYS\_TIME(time)

Converts UTC time to SYS time.

'time'	A time or list (qualified time) value or a vector of such values, UTC time(s).
Value:	A list (qualified time) or a vector of lists, SYS time(s).

This function either is equivalent to UTC\_TO\_LOCAL\_TIME or simply returns its argument (as qualified time value(s)), depending on the time reference of the system (SYS:BTR).

## 10.4.43 WEEK(time)]

The number of the week within a year.

The week numbering rule that is used may be given as the statement: "The 4th of January always belongs to week number 1".

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1 ... 53 or a vector of such integers.

## 10.4.44 YEAR(time)]

The year.

'time'	A time value or a vector of time values. Default value is the present SYS time.
Value:	An integer 1978 ... 2113 or a vector of such integers.

Examples:	
On 19th February 1997 at 20:35:04 o'clock the time functions returned the following values (supposing SYS:BTF == 0):	
Function call	Value
TIMES	"97-02-19 20:35:04"
TIMES(CLOCK-1)	"97-02-19 20:35:03"
TIME	"97-02-19 20:35"
TODS	"20:35:04"
DATE	"97-02-19"
YEAR	1997
MONTH	2
HOUR	20
MINUTE	35
SECOND	4
DOY	50
HOD	20.58444
HOY	1196.584
LOCAL_TIME_INFORMATION	If the time zone is +2 hours: LIST(UTC=...,DAYLIGHT_SAVING=FALSE, TIME_ZONE=2.0,BIAS=2.0,STATUS=0)
TIMES(PACK_TIME(1993,5,5,12,30,0))	"93-05-05 12:30:00"

## 10.5 String functions

The string functions perform operations on texts, bit strings and byte strings.

### 10.5.1 ASCII(n)

The Unicode character corresponding to the numeric character code.

'n'	An integer 0 ... 65535 or a vector of such integers.
Value:	A Unicode character or a vector of Unicode characters.

The name of this function has been reserved for compatibility although it now operates on Unicode characters.

See function ASCII\_CODE for the reverse operation.

Examples:

```
ASCII(65)           ; returns "A"  
ASCII(13)          ; returns 'carriage return' character
```

## 10.5.2 ASCII\_CODE(c)

The numeric Unicode code of the character argument.

'c'	A Unicode character or a vector of Unicode characters.
Value:	An integer 0 ... 65535 or a vector of such integers.

The name of this function has been reserved for compatibility although it now operates on Unicode characters.

See function ASCII for the reverse operation.

Example:

```
ASCII_CODE("A") == 65
```

## 10.5.3 BCD\_TO\_INTEGER(bcd)

Converts BCD coded numbers to integers.

BCD (Binary Coded Decimal) values are represented in SCIL by values of BIT\_STRING data type. Each digit takes 4 bits (called a nibble), so the length of a BCD bit string is a multiple of 4. The length of a BCD value is limited to 9 digits.

'bcd'	Bit string value containing the BCD coded number.
Value:	List value with attributes
	STATUS Integer, status code of the conversion
	INT Integer, the result of conversion

If the bit string represents a valid BCD number, the STATUS attribute is set to OK\_STATUS (0) and the result of the conversion is returned in attribute INT.

If the bit string is invalid, i.e. it contains nibbles out of range 0 ... 9, its length is not a multiple of 4 or its length is greater than 36, STATUS is set to SCIL\_NOT\_A\_BCD\_NUMBER and INT attribute is not returned.

See function INTEGER\_TO\_BCD for the reverse operation.

Example:

```
#LOCAL bcd_string_of_number = INTEGER_TO_BCD(98)  
; returns list with BCD representation of the integer  
; as the value of the attribute named BCD  
#LOCAL l_of_number = BCD_TO_INTEGER(bcd_string_of_number.BCD)  
; returns list with attribute INT containing the  
; integer, 98  
#LOCAL i_number = l_of_number.INT  
; assign the value of the INT attribute to  
; the variable 'i_number'
```

## 10.5.4 **BIN(b)**

Represents bit strings and integers as text in binary format.

'b'	A bit string or integer, or a vector of bit strings and integers.
Value:	A text or text vector. The length of the binary representation of an integer is 32 characters, the length of the representation of a bit string is the same as the length of the bit string.

See function BIN\_SCAN for the reverse operation.

Example:

```
BIN(BIT_SCAN("010101")) == "010101"
BIN(23)                                ;returns "0000...0010111" (27 leading zeroes)
```

## 10.5.5 **BIN\_SCAN(string)**

Creates an integer or real value out of its binary text representation.

The text representation may contain leading blanks, a sign (+/-), digits 0 and 1 and a decimal point.

'string'	A text or a text vector.
Value:	An integer or a real value, or a vector of such values. If the text representation contains a decimal point, the result is a real value, otherwise it is an integer value. If the integer result falls outside the integer value range (see <a href="#">Section 5</a> ), the overflowed (high order) bits are discarded.

See function BIN for the reverse operation.

## 10.5.6 **BIT\_SCAN(string)**

Creates a bit string out of its text representation.

'string'	A text containing characters "0" and "1" only, or a vector of such texts.
Value:	A bit string or a vector of bit strings.

Example:

```
BIT_SCAN("101")                      ;creates a bit string of length 3
```

## 10.5.7 **CAPITALIZE(text)**

Capitalizes a text.

'text'	A text or a text vector.
Value:	A text or a text vector.

The function converts the first character of the text or each vector element to upper case and the rest of the text to lower case according to the Unicode standard..

See also functions LOWER\_CASE and UPPER\_CASE.

Example:

```
CAPITALIZE ("VÄSTERÅS") == "Västerås"
```

## 10.5.8 COLLECT(v, delimiter)

Collects text fields into a text.

The function collects the text fields given in a text vector into one text value, where the fields are delimited by the given delimiter character.

'v'	A text vector.
'delimiter'	A text value of length 1.
Value:	A text value.

See function SEPARATE for the reverse operation.

Example:

```
COLLECT(VECTOR ("Alpha", "Beta", "Gamma"), "+") == "Alpha+Beta+Gamma"
```

## 10.5.9 DEC(value [,length [,decimals]])

Represents integer and real values as text.

'value'	An integer or real value, or a vector of such values.
'length'	An integer 0 ... 65 535. The minimum length of the text representation. Default = 6.
'decimals'	An integer 0 ... 253 or a vector of such integers. The number of decimals included in the conversion of real or integer data to text. Default = 0. If the 'value' argument is a vector, this argument may be a vector of the same length. It defines the number of decimals for each element individually.
Value:	A text of 'length' characters or more, or a vector of such texts. If the number is shorter than 'length', the string is filled up by leading blanks.

See function DEC\_SCAN for the reverse operation.

Examples:

```
DEC(1000) == " 1000"  
DEC(-1,0) == "-1"  
DEC(1.3002,6,3) == " 1.300"
```

## 10.5.10 DEC\_SCAN(string)

Creates an integer or a real value out of its decimal text representation.

'string'	A text or a text vector. The text representation may contain leading blanks, a sign (+/-), digits and a decimal point. Any other character is considered as an error generating a bad status.
Value:	An integer or a real value, or a vector of such values. If the text representation contains a decimal point, the result is a real value, otherwise it is an integer value. If the integer result falls outside the integer value range (see <a href="#">Section 5</a> ), a real value is returned.

See function DEC for the reverse operation.

Examples:

```
DEC_SCAN("-5") == -5
DEC_SCAN("+40000.0") == 40000.0
```

## 10.5.11 EDIT(text, key)

Simple text editing.

The function removes spaces and tabs out of a text according to a specified rule.

'text'	A text or a text vector.
'key'	A text value. One of the following keywords:
"TRIM"	Removes leading and trailing spaces and tabs.
"LEFT_TRIM"	Removes leading spaces and tabs.
"RIGHT_TRIM"	Removes trailing spaces and tabs.
"COLLAPSE"	Removes all spaces and tabs.
"COMPRESS"	Replaces multiple spaces or tabs with single spaces.
Value:	A text or a text vector.

## 10.5.12 HEX(n)

Represents an integer as text in hexadecimal format.

'n'	An integer or a vector of integers.
Value:	A text of 4 or 8 characters or a vector of such texts. A 4-character text is returned if the value of the argument is in range -32 768 ... 32 767, otherwise an 8-character text is returned. Leading zeroes are used when necessary.

See function HEX\_SCAN for the reverse operation.

Example:

```
HEX(26) == "001A"
HEX(-1) == "FFFF"
HEX(123456) == "0001E240"
```

## 10.5.13 HEX\_SCAN(string)

Creates an integer or a real value out of its hexadecimal text representation.

The text representation may contain leading blanks, a sign (+/-), hexadecimal digits and a decimal point. Any other character is considered as an error generating a bad status. The allowed digits are 0 ... 9, A ... F and a ... f.

'string'	A text or a text vector.
Value:	An integer or a real value, or a vector of such values.
	If the text representation contains a decimal point, the result is a real value, otherwise it is an integer value. If the integer result falls outside the integer value range (see <a href="#">Section 5</a> ), the overflow (high order) bits are discarded.

See function HEX for the reverse operation.

Examples:

```
HEX_SCAN("F") == 15
HEX_SCAN("A.8") == 10.5
```

## 10.5.14 INTEGER\_TO\_BCD(int [,digits])

Represents an integer value as a BCD coded bit string

BCD (Binary Coded Decimal) values are represented in SCIL by values of BIT\_STRING data type. Each digit takes 4 bits so the length of a BCD bit string is a multiple of 4. The length of a BCD value is limited to 9 digits.

'int'	Integer value containing the integer to be converted.
'digits'	Integer value 0 to 9, number of BCD digits in the result Default is 0.
Value:	List value with attributes
	STATUS Integer status code of the conversion.
	BCD Bit string value, the BCD coded representation of 'int'.

If 'digits' is 0, a bit string long enough to hold the result is returned. Otherwise exactly 'digits' BCD digits are returned, padded with leading zeroes, if necessary.

If the value of 'int' can be represented as a BCD bit string (i.e. it is non-negative and in the range specified by 'digits'), the STATUS attribute is set to OK\_STATUS (0) and the result of the conversion is returned in attribute BCD.

If the argument is invalid, STATUS is set to SCIL\_ARGUMENT\_OUT\_OF\_RANGE and BCD attribute is not returned.

See function BCD\_TO\_INTEGER for the reverse operation.

Example:

```
#LOCAL i_A = 9876 #LOCAL l_bcd = INTEGER_TO_BCD(i_A,9)
;converts the value of the variable 'i_A'
;returns a list with attributes 'BCD' and 'STATUS'
#LOCAL t_bcd = bin(l_bcd.bcd)
;returns the BCD code converted to textformat
#LOCAL t_status = dec(l_bcd.STATUS)
;returns the status converted to textformat
#LOCAL lConverted = BCD_TO_INTEGER(l_bcd.bcd)
;returns a list with attributes 'INT' and 'STATUS'
#LOCAL i_Converted = lConverted.INT
;returns an integer as value of the attribute 'INT'
;from the list named 'lConverted'
#LOCAL t_Converted_Status = dec(lConverted.STATUS)
```

```
;returns the status converted to textformat
;from the list named 'lConverted'
```

## 10.5.15 JOIN(delimiter [, a]\*)

Joins text fields into a text.

The function joins the text fields given as arguments into one text value, where the fields are delimited by the given delimiter character.

'delimiter'	A text value of length 1.
'a'	Up to 31 text or text vector arguments.
Value:	A text value containing all non-blank text fields from the arguments 'a', delimited by 'delimiter' character.

See also functions COLLECT and SEPARATE.

Example:

```
JOIN("/", "A", VECTOR("B", "C", ""))
JOIN(".", PO:PIE10) ;Makes a display name for the object identifier of
;process object PO:P10, e.g.
"STA1.BAY2.DEV3"
```

## 10.5.16 LOCATE(string1, string2 [, "ALL"])

Locates a text string in a text.

'string1'	A text or text vector. The text to be searched.
'string2'	A text value. The text string to be located.
"ALL"	Text keyword. If given, LOCATE searches for all occurrences of 'string2' in 'string1'.
Value:	An integer ( $>=0$ ), a vector of such integers or a vector of vectors of such integers. It represents the start position(s) of 'string2' in 'string1'. Zero result means that the string was not found.
Without "ALL":	The result represents the start position of the first found 'string2' in 'string1'. The result is an integer if 'string1' is a text value. If 'string1' is a vector, the result is a vector of the same length.
With "ALL":	The result represents the start positions of all occurrences of 'string2' in 'string1'. The result is a vector if 'string1' is a text value. If 'string1' is a vector, the result is a vector of the same length containing vectors of integers.

Example:

```
LOCATE ("ABC", "BC")
;returns 2
LOCATE ("ABC", "BC ")
;returns 0
LOCATE ("ABB", "B", "ALL")
;returns vector(2,3)
LOCATE (("FGHBBN", "ABBBB"), "BB", "ALL")
;returns (vector(4), vector(2,4))
LOCATE (("HBBN", "ABBBB", "BB", "AA"), "BB", "ALL")
;returns (vector(2), vector(2, 4), vector(1), vector(0))
```

## 10.5.17 LOWER\_CASE(text)

Converts text to lower case.

The function converts the upper case characters of 'text' to lower case according to the Unicode standard.

'text'	A text or a text vector.
Value:	A text or a text vector.

See also functions UPPER\_CASE and CAPITALIZE.

Example:

```
LOWER_CASE ("VÄSTERÅS") == "västerås"
```

## 10.5.18 OCT(n)

Represents an integer as text in octal format.

'n'	An integer or an integer vector.
Value:	A text of 6 or 11 characters or a vector of such texts. A 6-character text is returned if the value of the argument is in range -32 768 ... 32 767, otherwise an 11-character text is returned. Leading zeroes are used when necessary.

Examples:

```
OCT(10) == "000012"  
OCT(-1) == "177777"
```

## 10.5.19 OCT\_SCAN(string)

Creates an integer or a real value out of its octal text representation.

The text representation may contain leading blanks, a sign (+/-), digits of the radix and a decimal point. Any other character is considered as an error generating a bad status. The allowed digits are 0 ... 7.

'string'	A text or a text vector.
Value:	An integer or a real value, or a vector of such values. If the text representation contains a decimal point, the result is a real value, otherwise it is an integer value. If the integer result falls outside the integer value range (see <a href="#">Section 5</a> ), the overflowed (high order) bits are discarded.

See the OCT function for the reverse operation.

Examples:

```
OCT_SCAN("10") == 8  
OCT_SCAN("1.2") == 1.25
```

## 10.5.20 PACK\_STR(source, type [,length [,byte\_order]])

Creates a text, a bit string or a byte string out of its elements.

The function creates a text value out of a vector of its substrings, or a bit string or a byte string out of a vector of integers that represent the values of bit or byte fields of given length.

'source'	A vector of integer or text values.
'type'	Text keyword. The data type of the result: "TEXT", "BIT_STRING" or "BYTE_STRING".
'length'	An integer value, 1, 2, 4, 8, 16 or 32.  If 'type' = "BIT_STRING", this is the number of bits in each element of the source vector.  If 'type' = "TEXT", 'length' is ignored.
	If 'type' = "BYTE_STRING", the 'length' argument specifies how many bytes are initialized by each element of the vector. The length may be 1, 2 or 4. For example, if the length is 2, each element of the vector is taken as a 2-byte integer and then appended to the resulting byte string (The two high order bytes of the element are ignored).
	Default = 1.
'byte_order'	Text keyword: "BIG_ENDIAN" or "LITTLE_ENDIAN".  This argument is relevant only when 'type' is "BYTE_STRING". The 'byte_order' argument may be used to create byte strings for an application running in a computer of a different architecture. In a little endian architecture (for example, Intel, Alpha), integers are stored with the least significant byte first. In a big endian architecture (for example, Motorola), integers are stored with the most significant byte first. This argument should be used only if the resulting byte string is going to be exported into a foreign system requiring a specific byte ordering. If omitted, the byte string is created with the byte ordering of the underlying architecture.
Value:	Text, bit string or byte string depending on the 'type' argument.

See function UNPACK\_STR for the reverse operation.

Examples:

```
PACK_STR(("A", "B", "CD"), "TEXT") == "ABCD"
PACK_STR((0,1,0), "BIT_STRING") == BIT_SCAN("010")
PACK_STR((0,3,1), "BIT_STRING", 2) == BIT_SCAN("001110")
```

## 10.5.21 PAD([string, ]filler, length)

Pads a text with a filler string to the given length.

'string'	The text to be padded, default the empty string.
'filler'	The text to pad with.
'length'	Integer 0 ... 65 535, the wanted length of the result string.
Value:	Text, 'string' padded with 'filler' to the length 'length'.

If the length of 'string' is greater than 'length', the 'string' is returned as such.

Examples:

```
PAD("A", 5) == "AAAAA"
PAD("AA", "BC", 7) == "AABCBCB"
PAD(" ", MAX_TEXT_LENGTH); returns a blank text of the maximum length
PAD("ABC", "D", 2) == "ABC"
```

## 10.5.22 REPLACE(text, string, new\_string)

Replaces text strings by another string.

'text'	Text or text vector containing the input text.
'string'	Text value, the string to be replaced.
'new_string'	Text value, the replacing string.
Value:	Text or text vector containing the text in 'text' with all occurrences of 'string' replaced by 'new_string'.

The arguments 'string' and 'new\_string' may be of different length. Argument 'new\_string' may be empty. In this case, all the occurrences of 'string' are deleted.

If the resulting string is too long (> 65 535 characters), SCIL\_STRING\_TOO\_LONG error is generated when the 'text' argument is of text type. If it is a vector, the status of the overflowing element is set to SCIL\_STRING\_TOO\_LONG.

## 10.5.23 SEPARATE(text, delimiter)

Extracts fields of a text.

The function extracts the fields of text and returns the fields as a text vector. A field is any substring delimited by the given delimiter character.

'text'	A text value.
'delimiter'	A one character text value.
Value:	A text vector containing the fields of 'text'.
	The delimiter character itself is not included in the field. If no delimiter character is found in the text, the entire text is returned as the only element of the resulting vector. An empty field is returned in case of two consecutive delimiters.

See function COLLECT for the reverse operation.

Example:

```
SEPARATE ("A,B,C,D", ",")  
;returns the vector ("A","B","C","D").
```

## 10.5.24 SUBSTR(string, start [,length])

Extracts a substring from a text, bit string or byte string value.

'string'	A text, a bit string or a byte string value, or a vector of such values.
'start'	An integer 1 ... 65 535 (texts and bit strings) or 1 ... 8 388 600 (byte strings). The starting position of the substring.
'length'	An integer 0 ... 65 535 (texts and bit strings) or 0 ... 8 388 600 (byte strings). Default is 0. The length of the substring. If 'length' = 0, the function returns the end of 'string' starting from 'start'. If 'start' is greater than the length of 'string', an empty string is returned. If 'length' > 0 and the substring extends beyond the end of 'string', the result is padded with trailing blanks (texts), zero bits (bit strings) or zero bytes (byte strings).
Value:	The same data type as 'string'. The substring.

Examples:

```

SUBSTR ("ABCDE", 3, 2) == "CD"
SUBSTR ("ABCDE", 4, 3) == "DE "
SUBSTR ("ABCDE", 6, 3) == "    " ;3 blanks

V = BIT_SCAN("1010101")
BIN(SUBSTR(V,2,3)) == "010"

SUBSTR(BYTES,1,100000)
;returns a byte string containing the first 100000 bytes of byte string
BYTES.

```

## 10.5.25 UNPACK\_STR(source [,length [,byte\_order]])

Splits a text, a bit string or a byte string to a vector of its elements.

The function splits a text to a vector of substrings, or a bit string or a byte string into a vector of integers that represent the values of bit or byte fields of given length.

'source'	A text, bit string or byte string value.
'length'	Integer value. The length of the elements in the result vector.  If 'source' is a text, 'length' is the number of characters in each substring.
	If 'source' is a bit string, 'length' is the number of bits in each bit field. The following values are allowed: 1, 2, 4, 8, 16, 32. If the length of 'source' is not a multiple of 'length', the excess bits are ignored.
	If 'source' is a byte string, 'length' is the number of bytes in the byte fields. The values 1, 2 and 4 are allowed. 'length' may also be negative (-1, -2 or -4), in which case the integers in the byte string are taken as signed, otherwise they are taken as unsigned. If the length of 'source' is not a multiple of 'length', the excess bytes are ignored.
	Default = 1.
'byte_order'	Text keyword "BIG_ENDIAN" or "LITTLE_ENDIAN". The 'byte_order' argument specifies the byte ordering of the byte string (see PACK_STR above). It should only be used if the byte string has been imported from a foreign system. If omitted, the byte string is supposed to have the byte ordering of the underlying architecture.
Value:	A text or integer vector.

See function PACK\_STR for the reverse operation.

Examples:

```

UNPACK_STR("ABC") == ("A", "B", "C")
UNPACK_STR("ABCDE", 2) == ("AB", "CD", "E")
UNPACK_STR(BIT_SCAN("0111")) == (0,1,1,1)
UNPACK_STR(BIT_SCAN("011011"), 2) == (2,1,3)
UNPACK_STR(BIT_SCAN("0110111"), 2) == (2,1,3)

```

## 10.5.26 UPPER\_CASE(text)

Converts text to upper case.

The function converts the lower case characters of 'text' to upper case according to the Unicode standard.

'text'	A text or a text vector.
Value:	A text or a text vector.

See also functions LOWER\_CASE and CAPITALIZE.

Example:

```
UPPER_CASE("Västerås") == "VÄSTERÅS"
```

## 10.6 Bit functions

The bit functions manipulate integers and bit strings on bit level.

An integer consists of 32 bits. The bits are numbered 0 .. 31 from right to left, i.e., 0 = Least Significant Bit (LSB) and 31 = Most Significant Bit (MSB). A bit string can be composed of up to 65 535 bits numbered 1 ... 65 535 from left to right (see [Section 10](#)).

See also the SUBSTR function in [Section 10.5](#). This function can be used to extract a substring out of a bit string.

### 10.6.1 BIT(a, b)

The bit value of a given bit in a bit string or integer.

The function returns the bit value of bit number 'b' in 'a'.

'a'	An integer or a bit string value, or a vector of such values.
'b'	The bit number. An integer in the range 0 ... 31 or 1 ... 65 535 depending on the data type of 'a'.
Value:	Integer 0 or 1, or a vector of such integers.

Examples:

```
BIT(3,0) == 1
BIT(-1,15) == 1
BIT(BIT_SCAN("010101"),5) == 0
```

### 10.6.2 BIT\_AND(a1, a2)

Bitwise logical AND of the arguments.

'a1'	An integer or a bit string value, or a vector of such values.
'a2'	As 'a1'.
Value:	An integer or a bit string value, or a vector of such values.

If one of the arguments is a vector and the other is simple data, the operation is performed on the simple data and each vector element. If both arguments are vectors, the operation is performed on the corresponding elements. If the lengths of the vectors are unequal, odd elements are given SUSPICIOUS\_STATUS (see the Status Codes manual). Mixing integer and bit string values in one operation is not allowed.

Examples:

```
BIT_AND(6,5) == 4
BIT_AND(BIT_MASK(0,2,4), BIT_MASK(1,2,4)) == BIT_MASK(2,4)
```

### 10.6.3 BIT\_CLEAR(a [b]\*)

Sets given bits to 0.

The function calculates an integer by setting the bits numbered 'b' in 'a' to zero.

'a'	An integer or a bit string value, or a vector of such values.
'b'	The numbers of bits to set, in the range 0 ... 31 or 1 ... 65 535 depending on the data type of 'a'. Up to 31 bit numbers may be given. The bit numbers must not exceed the number of bits in 'a'.
Value:	The same data type as the argument 'a'.

Examples:

```
BIT_CLEAR(3,0) == 2
BIT_CLEAR(2,0) == 2
BIT_CLEAR(BIT_SCAN("01111"),2,4,5) == BIT_SCAN("00100")
```

## 10.6.4 BIT\_COMPL(a)

Logical bit complement of the argument.

The function calculates the bitwise logical NOT of the argument.

'a'	An integer or bit string value, or a vector of such values.
Value:	The same data type as the argument 'a'.

Examples:

```
BIT_COMPL(0) == -1
HEX(BIT_COMPL(HEX_SCAN("207F"))) == "DF80"
BIT_COMPL(BIT_SCAN("0101")) == BIT_SCAN("1010")
```

## 10.6.5 BIT\_MASK([b1 [,b]]\*)

Bit mask with given bits set to 1.

The function calculates an integer number by setting the given bits to 1 and all the others to 0.

'b1', 'b'	Bit numbers. Up to 32 integer values in the range 0 ... 31.
Value:	An integer.

Examples:

```
BIT_MASK() == 0
BIT_MASK(0) == 1
BIT_MASK(4,0) == 17
```

## 10.6.6 BIT\_OR(a1, a2)

Bitwise logical OR of the arguments.

'a1'	An integer or a bit string value, or a vector of such values.
'a2'	As 'a1'.
Value:	An integer or a bit string value, or a vector of such values.

If one of the arguments is a vector and the other is simple data, the operation is performed on the simple data and each vector element. If both arguments are vectors, the operation is

performed on the corresponding elements. If the lengths of the vectors are unequal, odd elements are given SUSPICIOUS\_STATUS (see the Status Codes manual). Mixing integer and bit string values in one operation is not allowed.

Examples:

```
BIT_OR(6,5) == 7
BIT_OR(BIT_MASK(0,2,4), BIT_MASK(1,2,4)) == BIT_MASK(0,1,2,4)
```

## 10.6.7 **BIT\_SET(a [,b]\*)**

Sets given bits to 1.

The function calculates an integer by setting the bits numbered 'b' in 'a' to one.

'a'	An integer or a bit string value, or a vector of such values.
'b'	The numbers of bits to set, in the range 0 ... 31 or 1 ... 65 535 depending on the data type of 'a'. Up to 31 bit numbers may be given. The bit numbers must not exceed the number of bits in 'a'.
Value:	The same data type as the argument 'a'.

Examples:

```
BIT_SET(0,3) == 8
BIT_SET(-1,15) == -1
BIT_SET(BIT_SCAN("0101"),1,3) == BIT_SCAN("1111")
```

## 10.6.8 **BIT\_STRING(length [,b]\*)**

Creates a bit string by setting given bits to 1 and the other ones to 0.

'length'	An integer, 1 ... 65 535. The number of bits in the bit string.
'b'	Up to 31 integer values in the range 1 ... 'length'. The numbers of the bits to be set to 1.
Value:	Bit string.

Example:

```
BIT_STRING(5,1,3,5) == BIT_SCAN("10101")
```

## 10.6.9 **BIT\_XOR(a1, a2)**

Bitwise logical XOR (exclusive OR) of the arguments.

'a1'	An integer or a bit string value, or a vector of such values.
'a2'	As 'a1'.
Value:	An integer or a bit string value, or a vector of such values.

If one of the arguments is a vector and the other is simple data, the operation is performed on the simple data and each vector element. If both arguments are vectors, the operation is performed on the corresponding elements. If the lengths of the vectors are unequal, odd elements are given SUSPICIOUS\_STATUS (see the Status Codes manual). Mixing integer and bit string values in one operation is not allowed.

Examples:

```
BIT_XOR(6,5) == 3
BIT_XOR(BIT_MASK(0,2,4), BIT_MASK(1,2,4)) == BIT_MASK(0,1)
```

## 10.7 Vector handling functions

The vector functions perform various operations on vector data.

### 10.7.1 APPEND(v, data)

Appends data to a vector.

The function creates a vector by appending 'data' to the contents of vector 'v'. If 'data' is a vector, the elements of 'data' are appended to 'v' resulting to a vector whose length is the sum of the argument vectors. If 'data' is of any other data type, the length of the resulting vector is one greater than the length of 'v' and the last element gets the contents of 'data'.

'v'	A vector.
'data'	Any SCIL data type.
Value:	A vector.

Example:

```
V1 = (1,2,3)
V2 = (4,5)
APPEND(V1,V2)           ; returns (1,2,3,4,5)
APPEND(V1,6)            ; returns (1,2,3,6)
```

### 10.7.2 BINARY\_SEARCH(v, value)

Searches an ordered vector by its element contents.

'v'	The vector to be searched.
	The elements of the vector may be of integer, real, time, text or Boolean type.
'value'	The value to be searched for.
	Data types allowed: integer, real, time, text or Boolean.
Value:	Integer, the index of the occurrence of 'value' in the vector. Zero will be returned, if 'value' is not found.

The vector 'v' must be arranged in ascending or descending order. Its elements must be all numeric (integer or real), all time values, all text values or all Boolean values.

If the vector contains more than one element by value 'value', the function returns the index of any of them.

The validity of vector 'v' is not checked. If the vector is not arranged, the function may or may not find the value. If the data type rules are violated, the function may fail by status SCIL\_INCOMPATIBLE\_TYPES, or it may find the requested value or return a zero.

To search an unordered vector by its contents, see function FIND\_ELEMENT. When the vector is long, BINARY\_SEARCH is much faster than FIND\_ELEMENT and should therefore be used whenever possible.

Example:

```
V = ("AB", "CD", "CD", "EF")
I = BINARY_SEARCH(V, "EF")           ; returns 4
I = BINARY_SEARCH(V, "CD")           ; returns 2 or 3
I = BINARY_SEARCH(V, "ab")           ; returns 0 (the search is case-
                                         sensitive)
```

### 10.7.3 CLASSIFY(v, n, low, high)

Classifies the elements of a vector into size classes and returns the counts of each class.

'v'	A vector with real elements. The vector to be classified.
'n'	An integer in the range 1 ... 2 000 000. The number of classes.
'low', 'high'	Integer or real numbers ('high' > 'low').
Value:	A vector of length 'n' with real elements. The number of elements in each class.

The range 'low' .. 'high' is divided into 'n' size classes of equal length ('high' - 'low')/'n'. The function calculates the count of elements of 'v' in each class. If an element in 'v' is smaller than or equal to 'low', it is counted to the lowest class. Elements larger than or equal to 'high' are counted to the highest class. An element on a class boundary is classified to the upper class.

This function is frequently used when calculating duration curves.

Example:

```
A = (1.0, 5.0, 3.0)
B = CLASSIFY(A, 2, 0.0, 10.0)
;The contents of B:
;B(1) == 2.0 (the range 0.0 .. 5.0)
;B(2) == 1.0 (the range 5.0 .. 10.0)
```

### 10.7.4 CUMULATE(v)

Accumulates the elements of the argument vector.

Each element n of the result vector is set to the sum of the n first elements of the argument vector.

'v'	A vector with real elements.
Value:	A vector of the same length as the argument vector. The elements of the vector are of real type.

The function uses double precision (64-bit) floating point arithmetics internally to achieve the best possible accuracy.

This function is frequently used when calculating duration curves.

Example:

```
V = (1, -5.6, 3.3, 37)
CUMULATE(V)                      ;returns (1.0, -4.6, -1.3, 35.7)
```

### 10.7.5 DELETE\_ELEMENT(v, index [,index2])

Deletes individual elements of a vector.

'v'	Any vector expression.
'index'	A positive integer or an integer vector specifying the elements to be deleted. If index is higher than the length of vector 'v' or negative, nothing is deleted.
'index2'	A positive integer ( $\geq$ index). If given, the range index .. index2 is deleted. If omitted, only the element(s) specified by 'index' are deleted.
Value:	A vector which is otherwise identical to 'v', but the element(s) specified by the index or index range are deleted.

Example:

```
V = (1,3,5,7,9)
I = (6,2,4,0,4)
R = DELETE_ELEMENT(V,I) ;returns vector (1,5,9)
```

## 10.7.6 FIND\_ELEMENT(v, value [,start\_index [,case\_policy]])

Searches a vector by its element contents.

'v'	A vector of any element type.
'value'	Value to be searched for, may be of any type.
'start_index'	Positive integer, default is 1. The element index to start search.
'case_policy'	Text keyword "CASE_SENSITIVE" or "CASE_INSENSITIVE", default is "CASE_SENSITIVE". Meaningful only if the data type of 'value' is TEXT.
Value:	Integer, the index of the first occurrence of 'value' in the vector, or the first occurrence at or after 'start_index'. Zero is returned, if 'value' is not found.

To search an ordered (ascending or descending) vector by its contents, see function BINARY\_SEARCH. When the vector is long, BINARY\_SEARCH is much faster than FIND\_ELEMENT and should therefore be used whenever possible.

Example:

```
V = ("AB", "CD", "EF", "CD")
I = FIND_ELEMENT(V, "CD")           ;returns 2
I = FIND_ELEMENT(V, "CD", I + 1)    ;returns 4 (second occurrence of
I = FIND_ELEMENT(V, "CD", I + 1)    "CD") ;returns 0 (no more found)
```

## 10.7.7 HIGH(v), LOW(v)

The largest (HIGH) or the smallest (LOW) element in a vector.

'v'	A vector with elements of integer, real or time type.
Value:	A vector of one element which is the largest or smallest element of the argument vector.
	If all the elements of the argument vector are of the same data type, the resulting data type will follow. Otherwise, a real type result is returned.

The result is returned as a one-element vector to be able to return the status of the result: The status of the only element is set to SUSPICIOUS\_STATUS (= 1), if any of the elements of the argument vector has a bad status (other than 0 or 3). If the argument vector is empty, or it contains no valid elements, NOT\_SAMPLED\_STATUS is set.

Example:

```
V = (1,-5.6,3.3,37)
HIGH(V) == 37.0
LOW(V) == -5.6
```

## 10.7.8 HIGH\_INDEX(v), LOW\_INDEX(v)

The index of the largest or smallest element in a vector.

'v'	A vector with elements of integer, real or time type.
Value:	An integer.

HIGH\_INDEX returns the index of the largest and LOW\_INDEX the index of the smallest element in the argument vector. If the argument vector is empty, or it contains no valid elements, the function returns the value 0.

Examples:

```
V = (1,-5.6,3.3,37)
HIGH_INDEX(V) == 4
LOW_INDEX(V) == 2
```

## 10.7.9 INSERT\_ELEMENT(v, pos, contents)

Inserts new elements into a vector.

'v'	Any vector.
'pos'	Integer 1 ... 2 000 000 or a vector of such integers. The position(s) where to insert new elements.
'contents'	Any type value. The value(s) assigned to inserted element(s).
Value:	A vector combined from the elements of 'v' and inserted new elements.

Position 'pos' (or each element of it, if a vector) is given as the index of the element in vector 'v' that is to succeed the inserted element in the result vector. As an example, if 'pos' is 1, the new element(s) are inserted at the beginning of vector. If 'pos' specifies an index that is larger than the length of 'v', the vector is expanded and elements that are not assigned a value will have status 10 (NOT\_SAMPLED\_STATUS).

The allowed combinations of 'pos' and 'contents' types are the following:

1. 'pos' is an integer
  - 1.1. 'contents' is not a vector A new element with contents 'contents' is inserted to become element 'pos' in the result vector.
  - 1.2. 'contents' is a vector New elements with contents specified by elements of 'contents' are inserted in the position specified by 'pos'.
2. 'pos' is an integer vector
  - 2.1. 'contents' is not a vector New elements are inserted in positions specified by elements of 'pos'. All new elements are assigned the value 'contents'.
  - 2.2. 'contents' is a vector The lengths of vectors 'pos' and 'contents' must be equal. Each contents(i) is inserted in position pos(i). If same index appears more than once in vector 'pos', the new elements are inserted in the order they appear in 'contents'.

Examples:

Suppose V is a vector with contents (1,2). Then

```

INSERT_ELEMENT(V, 1,          ; returns (0,1,2)
INSERT_ELEMENT(V, 2, ("A","B")) ; returns (1,"A","B",2)
INSERT_ELEMENT(V, 4, 4)        ; returns (1,2,'bad status 10',4)
INSERT_ELEMENT(V, (1,1,2,3), 0) ; returns (0,0,1,0,2,0)
INSERT_ELEMENT(V, (1,2,2),     ; returns ("A",1,"B","C",2)
("A","B","C"))

```

## 10.7.10 INTERP(v, x)

Interpolates a value from a curve.

'v'	A vector with real type elements.
'x'	A real value.

The argument 'v' is interpreted as (x, y) coordinate pairs defining a curve. The 1st and 2nd element define the first point (x, y) of the curve, the 3rd and 4th define the second point, etc. The x coordinates must be given in ascending order. The y coordinate corresponding to the given 'x' is interpolated from the curve and returned as a real number. If 'x' < x1, y1 is returned. If 'x' > xn, yn is returned (where 'n' denotes the number of coordinate pairs in the vector).

Value: A real number.

Examples:

```

A(1) = 1.0          ;X1
A(2) = 6.0          ;Y1
A(3) = 3.0          ;X2
A(4) = 7.0          ;Y2
INTERP(A,1.0) == 6.0
INTERP(A,2.0) == 6.5
INTERP(A,5.0) == 7.0

```

## 10.7.11 INVERSE(v, n, low, high)

Inverts a curve.

The elements in 'v' are interpreted to define a monotonously ascending curve  $y = y(x)$ , where the index of the vector represents the x-coordinate (1.0 .. length(v)) and the element value represents the y-coordinate. The INVERSE function inverts this curve, i.e. solves the curve  $x = x(y)$  by linear interpolation.

'v'	A vector with real elements given in ascending order.
'n'	An integer in the range 1 ... 2 000 000, the length of the resulting vector.
'low', 'high'	Real numbers (high > low).
Value:	A vector of length 'n' with real elements.

The numeric solution is returned in the resulting vector elements so that the i:th element gives the x-coordinate corresponding to  $y = \text{low} + i * (\text{high} - \text{low}) / n$ . If  $y < v(1)$ , value 1.0 is assigned. If  $y > v(n)$ , where n is the length of 'v', value n is assigned.

This function is frequently used when calculating duration curves.

Examples:

```

A(1) = 3.0
A(2) = 4.0
A(3) = 9.0

X = INVERSE(A,5,0.0,10.0)
;The vector X now consists of the elements:
; 1.0   (y=2)
; 2.0   (y=4)
; 2.4   (y=6)
; 2.8   (y=8)
; 3.0   (y=10)

```

For y=2, x gets the value 1.0, because this is the lowest possible value, and for y=10, x gets the value 3.0, because this is the highest possible value.

## 10.7.12 MEAN(v)

The mean value of the elements of a vector.

This function calculates the sum of all valid elements of a vector and divides the sum by their count.

'v'	A vector with real elements.
Value:	A vector of one element. The data type of the element is real.

The result is returned as a one-element vector to be able to return the status of the result: The status of the only element is set to SUSPICIOUS\_STATUS (=1), if any of the elements of the argument vector has a bad status (other than 0 or 3). If the argument vector is empty, or it contains no valid elements, NOT\_SAMPLED\_STATUS is set.

The function uses double precision (64-bit) floating point arithmetics internally to achieve the best possible accuracy.

Example:

```
V = (1,-5.6,3.3,37)
M = MEAN(V)           ;Now M(1) == 8.925
```

## 10.7.13 PICK(v, indices)

Picks up specified elements from a vector.

'v'	Any vector value.
'indices'	An integer vector with elements in the range 1 ... 2 000 000. The vector specifies the indices of the elements which will be selected from the source vector.
Value:	A vector, of the same length as 'indices', containing the elements selected from the source vector.

If the index vector contains indices that are not present in the source vector (i.e. indices <= 0 or indices that are greater than the length of 'v'), the corresponding elements in the result vector are given NOT\_SAMPLED\_STATUS.

This function is frequently used in conjunction with functions SELECT, SORT and SHUFFLE.

Example:

```
#LOCAL P, N, S
P = (1,2,3,5,7,11,13,17,19)
N = (7,4,1)
```

```
S = PICK(P,N)
;Now S == (13,5,1)
```

## 10.7.14 REMOVE\_DUPLICATES(v [,status\_handling [,case\_policy]])

Removes duplicate elements of a vector.

'v'	Vector value to be examined.
'status_handling'	Text keyword "CONSIDER_STATUS" or "IGNORE_STATUS", default = "IGNORE_STATUS"
'case_policy'	Text keyword value, either "CASE_SENSITIVE" or "CASE_INSENSITIVE", default = "CASE_SENSITIVE"
Value:	Vector containing the different element values of 'v'.

The arguments 'status\_handling' and 'case\_policy' may be given in any order.

Two elements are considered equal if all the following conditions are satisfied:

- The value types are the same.
- The values are the same.
- The status values are the same (when "CONSIDER\_STATUS") or else both valid (<= LAST\_VALID\_STATUS) (when "IGNORE\_STATUS").

Text values are compared for equality according to the argument 'case\_policy'.

The test for equality is recursive, i.e. if an element is a vector or list, its components are tested for equality.

The element values are returned in the order of appearance in 'v'.

Example:

```
REMOVE_DUPLICATES(VECTOR("ABC", "abc", "DEF"), "case_insensitive")
;returns VECTOR("ABC", "DEF")
```

## 10.7.15 REVERSE(v)

Reverses the order of elements of a vector.

'v'	Any vector.
Value:	A vector of the same length as the argument vector.

Example:

```
V = (1,-5.6,3.3,37)
REVERSE(V)
;returns (37,3.3,-5.6,1)
```

## 10.7.16 SELECT(source, condition [,wildcards])

Selects the elements of a vector or a list of vectors that fulfil given condition.

'source'	A vector or a list of vector attributes of equal length.	
'condition'	A text containing the selection criterion. The syntax depends on the data type of the source, see below. The condition is evaluated in read-only mode.	
'wildcards'	Text keyword, default "NO_WILDCARDS".	
	"WILDCARDS"	Wildcards are used.
	"NO_WILDCARDS"	Wildcards are not used.
Value:	An integer vector containing the selected element indices.	

If 'source' is a vector, each element of the vector is matched for the given condition. The indices of the matching elements are returned in the result vector. The syntax of the condition is the normal SCIL expression syntax (see [Section 4](#)), but the value of the element is referred to by notation `()`. Even the parentheses may be omitted, when the element is used as the left hand operand of a relation. The expression must evaluate to a boolean value. Local variables and arguments of the program may not be used as operands in the condition. Examples of valid conditions:

">= 1 AND <= 10"	;Selects elements in range 1 .. 10	
"() >= 1 AND () >= 10"	;Same as above	
;Selects all odd elements and the ones > 10		
"==" "A*****"	;Selects elements starting with letter A	

If 'source' is a list, the attribute values of each index are matched for the given condition. The syntax of the condition is the normal SCIL expression syntax, but the value of the attribute element is referred to by the attribute name. Examples of valid conditions:

"AB >= 1 AND CD <= 10"	;Selects all i:	source.AB(i) >= 1 and source.CD(i) <= 10
"ABC > 10 OR ODD(EFG)"	;Selects all i:	source.ABC(i) > 10 and source.EFG(i) is odd
"LN == ""A*****" AND OV == 1"	;Selects all i:	source.LN(i) starts with A and source.OV(i) == 1

In text strings given in argument 'condition', wildcard character % can be used to represent any single character and \* to represent any sequence of characters. For example, "%B\*C" matches with any string that has letter B as its second character and letter C as its last character. To use wildcards, "WILDCARDS" argument must be given, otherwise characters % and \* have no special meaning.

Because SELECT returns the indices of the selected elements, function PICK is frequently applied to the result to get the element values.

Examples:

```
@A = (1.0,2.5,7.0,10.6)
@I = SELECT(%A,">= 2 AND < 10")
;Now I == (2,3)
@S = SELECT(A:DOS(1..30),">0")
@A = PICK(B:DOV(1..30),%S)
;The registrations of the data object A that have a status >0 are
selected and
;the corresponding values of the data object B are picked.
;If 'source' is a list containing the attributes AB and CD the condition
can be written, ;e.g., ;"(AB > 10) AND (CD == 5)" which implies that the
indices which
;fulfil these conditions are selected.
```

```
SELECT(%V, "GET_STATUS () == 0")
;means that the elements with status == 0 in the vector %V are selected.
```

## 10.7.17 SHUFFLE(n)

Shuffles integers 1 to n into a random order.

The function creates a vector of length n that contains integer values 1 to n in a random order.

'n'	Integer value in range 0 ... 2 000 000.
Value:	Integer vector of length 'n'.

Example:

To rearrange any vector V to a random order:

```
SHUFFLED = PICK(V, SHUFFLE(LENGTH(V)))
```

## 10.7.18 SORT(v, [start, [length]])

Sorts a vector.

The function sorts a vector of numeric (integer, real or time) or text data into ascending order or alphabetical order (ASCII code order), respectively. When text data is sorted, a substring of the text may be selected to be used as the sort key.

'v'	The vector to be sorted. The elements of the vector must be uniform, either numeric or text.
'start'	The start position (1 ... 65 535) of the sort key when text data is sorted. Default = 1, i.e., the first character of the text.
'length'	The length of the sort key (1 ... 65 535). If omitted, the substring from start position to the end of text is used as the key.
Value:	An integer vector.  If the length of 'v' is n, the length of the resulting vector is also n and it contains integers 1 to n in the collating sequence of the elements of 'v'. For instance, if the first element of the result vector is 25, the 25th element of the 'v' vector is the 'smallest' one.

If descending order is required, apply function REVERSE to the result of SORT. Function PICK is frequently used to physically rearrange the elements into the sort order.

Example:

```
#LOCAL UNSORTED = (7, 9, 4, 3, 35, 6)
#LOCAL INDEX = SORT(UNSORTED)

;Now: INDEX == (4, 3, 6, 1, 2, 5)
SORTED = PICK(UNSORTED, INDEX)
```

## 10.7.19 SPREAD(v, indices, new\_value)

Replaces vector elements by a new value.

'v'	Any vector. The source vector.
'indices'	A vector of integer elements, 1 ... 2 000 000. The index vector containing the indices to be replaced.
'new_value'	Any SCIL data type. The replacing value(s).
Value:	A vector of the same length as 'v'.

The function creates a vector that is otherwise identical to the argument vector, but the elements specified by an index vector are replaced by a new value.

If 'new\_value' is of simple (non-vector) data type, the elements listed by 'indices' are replaced by that value. If 'new\_value' is a vector, its elements replace the elements listed the index vector. If vectors 'indices' and 'new\_value' are unequal in length, the extra elements of the longer vector are disregarded. indices in the index vector that are greater than the length of 'v' are disregarded.

Example:

```
#LOCAL V, I, A, S, T
V = (1,2,3,4,5)
I = (1,3,5)
A = (6,7,8)
S = SPREAD(V,I,A)
T = SPREAD(V,I,0)
;Now
;S == (6,2,7,4,8)
;T == (0,2,0,4,0)
```

## 10.7.20 SUM(v), SUM\_POS(v), SUM\_NEG(v)

The sum of all or the positive or the negative elements of a vector.

The functions calculate the sum of all (SUM), positive (SUM\_POS) or negative (SUM\_NEG) valid elements of a vector, respectively.

'v'	A vector with integer or real or elements.
Value:	A vector of one element. The element is an integer if all the elements of 'v' are integers, otherwise it is a real number.

The result is returned as a one-element vector to be able to return the status of the result: The status of the only element is set to SUSPICIOUS\_STATUS (= 1), if any of the elements of the argument vector has a bad status (other than 0 or 3). If the argument vector is empty, or it contains no valid elements, NOT\_SAMPLED\_STATUS is set.

The functions use double precision (64-bit) floating point arithmetics internally to achieve the best possible accuracy.

Examples:

```
#LOCAL V, POS, NEG
V = (1,-5.6,3.3,37)
POS = SUM_POS(V)
NEG = SUM_NEG(V)
;Now
;POS(1) == 41.3
;NEG(1) == -5.6
```

## 10.7.21 TREND(v, n)

Returns the last ('newest') elements of a vector.

'v'	Any vector.
'n'	An integer in the range 0 ... 2 000 000. The number of elements to be included in the result vector.
Value:	A vector of length 'n'. If 'n' is greater than the length of the argument vector, the first elements get no value, their status is set to NOT_SAMPLED_STATUS.

Examples:

```
#LOCAL V = (1,-5.6,3.3,37)
TREND(V,2)
;returns vector (3.3,37)
```

## 10.7.22 VECTOR ([element1 [,element]\*])

Creates a vector out of given elements.

'element'	An expression of any data type. Up to 2 000 000 elements can be given. Using VECTOR without an argument list creates an empty vector.
Value:	A vector with the given elements.

Example:

```
#LOCAL A = VECTOR()
;The variable A will be an empty vector.
```

# 10.8 List handling functions

List handling functions do various operations on list value data.

## 10.8.1 ATTRIBUTE\_EXISTS(list, attribute)

Checks whether a list contains given attribute(s).

'list'	Any list value.
'attribute'	Text or text vector, the name(s) of attribute(s).
Value:	A boolean or boolean vector. TRUE if the given attribute is found in the list, otherwise FALSE.

## 10.8.2 DELETE\_ATTRIBUTE(list, attribute)

Deletes attribute(s) from a list.

'list'	Any list value.
'attribute'	Text or text vector, the name(s) of attribute(s) to be deleted.
Value:	A list, which is a copy of 'list' but attribute(s) specified by 'attribute' are removed.

Attributes that do not exist in 'list' are silently ignored, no error is generated.

Example:

The following statement reads the definition of a process object and removes attributes LN and IX out of it.

```
#LOCAL A = DELETE_ATTRIBUTE(FETCH(0, "P", "ABC", 1), ("LN", "IX"))
```

### 10.8.3 LIST([attribute = expression, [attribute = expression]\*])

List created out of given attribute name/value pairs.

The argument list may contain up to 2 000 000 attribute assignments.

'attribute'	A freely chosen attribute name of up to 63 characters.
'expression'	An expression of any data type. The value assigned to the 'attribute'.
Value:	A list of the given attributes.

Syntactically, LIST is actually not a function, because it does not have expressions as its arguments.

Example:

```
#LOCAL STUFF = LIST(NUMBERS = (1, 2 ,3), NAMES = ("A", "B", "C"))
```

### 10.8.4 LIST\_ATTR(list)

Names of attributes of a list.

'list'	A list value.
Value:	A text vector containing the names of attributes of 'list' in alphabetical order.

Example:

```
#LOCAL X = LIST(AA = 1, BB = 2, CM = "TEST")
LIST_ATTR(X) ;returns VECTOR("AA", "BB", "CM")
```

### 10.8.5 MERGE\_ATTRIBUTES(left, right)

Merges two lists into one.

'left'	Any list value.
'right'	Any list value.
Value:	A list which contains all the attributes of 'left' and 'right' lists.
	If the same attribute exists in both 'left' and 'right', the value in 'right' is returned.

Example:

```
#LOCAL X = LIST(AA = 1, BB = 2, CM = "TEST")
LIST_ATTR(X) ;returns VECTOR("AA", "BB", "CM")
```

```

#LOCAL X = LIST(A = 1,B = 2)
#LOCAL Y = LIST(C = 3,D = 4)
#LOCAL Z
X = MERGE_ATTRIBUTES(X, LIST(C      ;X contains LIST(A = 1,B = 2,C = 4)
= 4))                                ;Z contains LIST(A = 1,B = 2,C =
Z = MERGE_ATTRIBUTES(X, Y)            3,D = 4)

```

## 10.9 Functions related to program execution

### 10.9.1 ARGUMENT(n)

Nth argument of the program call.

'n'	Integer value 1 ... 32.
Value:	The value of the nth argument of the argument list. If there are fewer than 'n' arguments in the list, a value with data type "NONE" is returned.

For efficiency and clarity, it is recommended to name the arguments using the #ARGUMENT command. However, if the SCIL program is designed to take a varying number of arguments, this function along with ARGUMENT\_COUNT is frequently used to read the optional arguments.

Example:

```

#ARGUMENT A
#LOCAL B = 1                         ;Default value for the second argument
#if ARGUMENT_COUNT == 2 #THEN B = ARGUMENT(2)

```

### 10.9.2 ARGUMENT\_COUNT

The total number of arguments of the program call.

Value:	Integer, 0 ... 32.
--------	--------------------

### 10.9.3 ARGUMENTS

All arguments of the program call as a vector.

Value:	Vector with up to 32 elements. If the program call has no arguments, a zero length vector is returned.
--------	--

Example:

```

@A = ARGUMENTS
@B = DO (%PROGRAM, %A)

```

### 10.9.4 DO(program [,a]\*)

Executes the SCIL program given as an argument.

'program'	A text vector containing the SCIL program to be executed.
'a'	Any SCIL data type. These arguments are passed to the SCIL program (up to 31 arguments may be specified).
Value:	The value returned by the #RETURN command in the executed program, or 0 if the program did not terminate by #RETURN command.

This function is recommended instead of #DO command, because arguments and return values are not supported by #DO command.

Example:

```
;An example that calculates the tangent function of its argument.  
#LOCAL RESULT = DO(READ_TEXT("TANGENT.SCL"), 0.5)  
  
;Contents of file TANGENT.SCL:  
#ARGUMENT ANGLE  
#LOCAL COSA = COS(ANGLE)  
#IF COSA == 0.0 #THEN #ERROR RAISE  
STATUS_CODE("SCIL_ARGUMENT_OUT_OF_RANGE")  
#ELSE #RETURN SIN(ANGLE) / COSA
```

## 10.9.5 **ERROR\_STATE**

Returns the current error handling policy.

Value:	Text value depicting the current error handling policy: "STOP", "CONTINUE", "IGNORE" or "EVENT".
--------	--

Example:

```
#LOCAL PREVIOUS_ERROR_STATE  
PREVIOUS_ERROR_STATE = ERROR_STATE  
#ERROR IGNORE  
; Run ignoring errors  
...  
#ERROR 'PREVIOUS_ERROR_STATE'
```

## 10.9.6 **MEMORY\_USAGE(keyword, arg)**

The amount of pool memory allocated for the argument.

'keyword'	Text keyword, "EXPRESSION" or "VARIABLE".
'arg'	Any expression, if 'keyword' = "EXPRESSION".
Value	A text or text vector, the name(s) of the variable(s), if 'keyword' = "VARIABLE".

Integer or integer vector, the amount of memory pool allocated for the expression or the variable(s).

This function helps debugging and analysing SCIL applications. It returns the amount of memory pool (as bytes) allocated for an expression, or for a variable (including the book keeping data and the value of the variable).

Examples:

(Memory usage in current implementation of SCIL, may change in future releases.)

```
MEMORY_USAGE ("EXPRESSION", "ABC") == 8  
MEMORY_USAGE ("EXPRESSION", 1) == 0 ;since integer values are not allocated
```

```
;from the pool
@X = "ABC"
MEMORY_USAGE ("VARIABLE", "X") == 136
```

## 10.9.7 OPS\_CALL(command [,nowait]), OPS\_CALL(command [,option1 [,option2]])

Executes an operating system command.

'command'	A text value. The command to be executed. The maximum length of command is 32 767 characters. The started application may limit the accepted length from this value.
'nowait'	Integer 0 ... 16. If omitted, OPS_CALL starts the execution of the command and waits until it has finished. If given, termination is not waited. The value of the argument has no meaning in current implementation. The allowed range is preserved for compatibility.
'options'	One or two keywords in any order:
	"WAIT" OPS_CALL waits until the command has finished.
	"NOWAIT" OPS_CALL does not wait the command termination. This is the default value.
	"CLIENT" The command is executed in the caller's session. This is the default value. See below.
	"SERVER" The command is executed in the server session. See below.
Value:	A list with the following attributes:
	ST Integer, the status value returned by the operating system. 0 = OK, any other value = failure.
	FN 0. The attribute has no meaning in current implementation. It is preserved for compatibility.

The option argument values "CLIENT" and "SERVER" are ignored in systems running an operating system older than Windows Vista or Windows Server 2008. The following applies to newer operating systems only.

The option argument "SERVER" specifies that the command is to be executed in the session of the system server (Windows session #0, where the MicroSCADA service is running). The value "CLIENT" specifies that the command is to be executed in the caller's session. If the caller runs in the server session, the command is executed in the server session regardless of the argument value. The default value for the argument is "CLIENT".

When OPS\_CALL is executed in a client session (other than session 0), the started program has access to the caller's desktop (if any), i.e. it may have a visible window and it may receive user input. When the user logs out, any process started by OPS\_CALL without "SERVER" argument is terminated by the operating system.

When OPS\_CALL is executed in the server session, the started program has no access to a desktop. The program runs until it terminates itself or the MicroSCADA service is stopped.

## 10.9.8 OPS\_PROCESS(command [,directory [,option1 [,option2]]])

Starts an external program as a separate process.

'command'	Text value containing the command to start the process, for example, "\tools\my_tool my_file -my_option". The maximum length of command is 32 767 characters. The started application may limit the accepted length from this value.
'directory'	Optional text value containing the work or default directory to be used by the process. The directory is given in OS dependent format (see PARSE_FILE_NAME function to obtain OS directory names). An empty name, "", denotes the work directory of the caller. Default value = "".
'options'	One or two keywords in any order:  "WAIT" OPS_PROCESS waits until the command has finished. "NOWAIT" OPS_PROCESS does not wait the command termination. This is the default value. "CLIENT" The command is executed in the caller's session. This is the default value. See below. "SERVER" The command is executed in the server session. See below.
Value:	A list value containing the following attributes:  START_STATUS Integer value containing the OS dependent status code obtained when starting the process. EXIT_STATUS Integer value containing the OS and application dependent exit status of the process. This attribute is returned only if "WAIT" is specified and start of process succeeded.

The functionality is close to that of OPS\_CALL function. However, OPS\_PROCESS does not start the command interpreter, it simply runs the program given as an argument of the function call. In Windows this means that only EXE files may be started this way. To execute BAT files, OPS\_CALL should be used.

For the meaning of the option argument values "CLIENT" and "SERVER", see the description of the OPS\_CALL function.

## 10.9.9 REVISION\_COMPATIBILITY(issue [,enable])

Selects the compatibility issues to be used in the context.

'issue'	Text keyword. The name of the compatibility issue.
'enable'	An optional boolean argument. TRUE = compatibility is enabled. FALSE = compatibility is disabled. If this argument is not given, the function only reads the compatibility state without modifying it.
Value:	Boolean value indicating the compatibility before it was changed by the function.

The compatibility with the old revision is enabled/disabled by compatibility issues. The REVISION\_COMPATIBILITY function overrides temporarily, in the current SCIL context, the revision compatibility defined in the APL:BRC attribute. See the description of the APL:BRC attribute in the System Objects manual.

Example:

```
#LOCAL RC
RC = REVISION_COMPATIBILITY ("FILE_FUNCTIONS_CREATE_DIRECTORIES", FALSE)
; WRITE_TEXT function used here
RC = REVISION_COMPATIBILITY ("FILE_FUNCTIONS_CREATE_DIRECTORIES", RC)
```

The first function call disables the compatibility issue FILE\_FUNCTIONS\_CREATE\_DIRECTORIES. The second function call resets the compatibility issue to the value it had before.

## 10.9.10 STATUS

The status code of the last error in the program.

Value: A non-negative integer, status code. See the Status Codes manual.

The function reads and resets the internal 'last error' indicator.

**Example:**

```
#LOCAL S
#ERROR IGNORE
    S = STATUS
    #SET ABC:PBI = 0
    S = STATUS
#ERROR STOP
#IF S > 0 #THEN !SHOW ERROR S
#ELSE           !ERASE ERROR
```

## 10.9.11 VARIABLE\_NAMES

Lists the names of global variables defined in the SCIL context.

Value: A text vector, the names of global variables in alphabetic order.

# 10.10 Functions related to the run-time environment

## 10.10.1 AEP\_PROGRAMS(apl)

Lists the running Application Extension Programs of an application.

'apl' Integer value 0 ... 250, the logical application number. 0 = current application.

Value: A vector. Each element provides information about one AEP program invocation as a list with the following attributes:

PROGRAM_NUMBER	Integer, the AEP program number.
START_COMMAND	Text, the command that was used in AEP_START to start the program.
ARGUMENT	Any type, the SCIL argument given to the program when started.
START_TIME	Time value, the time program was started.
PROCESS_NAME	Text, the name of the executable,e.g. "TOPCAL.EXE".
PROCESS_ID	Integer, the process id (PID) of the executing process.

This function may be used to supervise the execution of external programs. See also [Section 10.10.4](#).

## 10.10.2 CONSOLE\_OUTPUT(text [,severity [,category]])

Writes a message into the system error log (SYS\_LOG.CSV) and into the notification window.

'text'	A text or text vector. A text vector generates a multiline message.
'severity'	Text, the severity of the message. One of the following values: " "      Unspecified. This is the default value. "I"      Information "W"      Warning "E"      Error "T"      Test "D"      Debugging "N"      Notification only (not written to SYS_LOG.CSV)
'category'	A free text, up to 4 characters. Use the name or abbreviation of the functionality producing the message. Default value "".
Value:	The status code of the operation. 0 = OK.

### 10.10.3 ENVIRONMENT(variable)

Retrieves an operating system environment variable value.

'variable'	A text specifying the name of the environment variable.
Value:	A text, the value of the environment variable. An empty text is returned if the environment variable does not exist.

In Windows, the names of available environment variables are shown by the Windows Command Shell command set (without any arguments).

Examples:

```
COMPUTER = ENVIRONMENT ("COMPUTERNAME")
WORKSTATION_COMPUTER = ENVIRONMENT ("CLIENTNAME")
```

### 10.10.4 IP\_PROGRAMS

Lists the running Integrated Programs in the system.

Value:	A vector. Each element provides information about one IP program invocation as a list with the following attributes:
PROGRAM_NUMBER	Integer, the IP program number.
START_COMMAND	Text, the command that was used in IP_START to start the program.
ARGUMENT	Any type, the SCIL argument given to the program when started.
START_TIME	Time value, the time program was started.
PROCESS_NAME	Text, the name of the executable, for example, "IP.EXE".
PROCESS_ID	Integer, the process id (PID) of the executing process.

This function may be used to supervise the execution of external programs. See also AEP\_PROGRAMS above.

### 10.10.5 MEMORY\_POOL\_USAGE(pool)

The amount of memory allocated from a memory pool.

'pool'	Text keyword specifying the pool to get information from:	
"GLOBAL"	The global memory pool.	
"PRIVATE"	The private memory pool of the thread.	
"LOCAL"	Obsolete argument value (former "Local pool of the process"). Allowed for compatibility, works as "PRIVATE".	
"PROCESS"	Obsolete argument value (former "Pool used by the process (either local or global)"). Allowed for compatibility, works as "PRIVATE" (however, see value attribute POOL below).	
Value:	List value with following attributes:	
SIZE	Current size of the pool as megabytes.	
KILOS	Current size of the pool as kilobytes. Non-zero only if SIZE is 0.	
MAX_POOL_SIZE	Maximum size of the pool to grow (megabytes).	
USED	Bytes used.	
FREE	Bytes free.	
BLOCK_SIZES	Vector of allocation block sizes used by the pool (only if pool size > 0).	
USED_BLOCKS	Vector counting allocations of each block size (only if pool size > 0).	
FREE_BLOCKS	Vector counting free blocks of each size (only if pool size > 0)	
POOL	Obsolete attribute, the value is always "LOCAL". Returned only if 'pool' is "PROCESS".	

## 10.10.6 OPS\_NAME([major [,minor]])

Returns the name of the operating system.

'major'	Integer value, the major version number.	
'minor'	Integer value, the minor version number, default value 0.	
Value:	Text value, the name of the operating system, whose version numbers match with 'major' and 'minor', see the table below.	

If the function is called without arguments, the name of the operating system that is running is returned.

The following table summarizes the values of the major and minor version numbers of current Windows versions that run SYS600:

Value	Major	Minor
"Windows"	4	0
"Windows 2000"	5	0
"Windows XP"	5	1
"Windows Server 2003"	5	2
"Windows Vista or Server 2008"	6	0
"Windows 7 or Server 2008 R2"	6	1

Table continues on next page

Value	Major	Minor
"Windows 8 or Server 2012"	6	2
"Windows 8.1 or Server 2012 R2"	6	3
"Windows 10, Server 2016 or Server 2019"	10	0

If the arguments do not match with any known operating system version, a question mark (?) is returned.

Tools that display operating system information are encouraged to use this function. By using it, they don't have to be updated when new operating system versions are taken into use.

## 10.10.7 **REGISTRY(function, key, value\_name)**

Reads the registry maintained by Windows operating system.

'function'	Text keyword value
	The only possible value is "READ" for now.
'key'	Text value, the key to be read.
'value_name'	Text value, the name of the value to be read.
	Empty string "" denotes the 'default' value.
Value:	A list with the following attributes:
	STATUS        Integer, the SCIL status code, 0 if OK.
	VALUE        Text or integer value, the value of the key.
	This attribute is returned only when STATUS = 0.

The function reads the value of the specified key from the HKEY\_LOCAL\_MACHINE section of the registry. Only value types REG\_SZ (text value) and REG\_DWORD (integer value) are supported. In case of a failure, one of the following status codes is returned in the STATUS attribute of resulting value:

SCIL\_REGISTRY\_KEY\_NOT\_FOUND, SCIL\_REGISTRY\_VALUE\_NOT\_FOUND,  
SCIL\_REGISTRY\_DATATYPE\_NOT\_SUPPORTED.

See operating system documentation for further information about registry keys, value names and value types.

Example:

```
#LOCAL RESULT
RESULT = REGISTRY("READ", "SOFTWARE\ABB\PAK
\SYS_500\MAIN_LICENSE", "CUSTOMER")
#IF RESULT.STATUS == 0 #THEN .DO_SOMETHING(RESULT.VALUE)
#ELSE_IF RESULT.STATUS == STATUS_CODE("SCIL_REGISTRY_KEY_NOT_FOUND")
#THEN -
    .MAIN_LICENSE_IS_MISSING
#ELSE .DO_SOMETHING_ELSE(RESULT.STATUS)
```

## 10.10.8 **SCIL\_HOST**

Returns the type and number of the process that is running this SCIL code.

For example, the start program of a picture can find out whether the picture is being displayed on a monitor or printed.

Value:	A list containing two attributes:
NAME	The process type as a text string. The possible values are: "MAIN" The main (start-up) process "PICO" A monitor process "REPR" A report process "PRIN" A printer spooler process processing a format picture "PRNC" A printer process processing a page header "EXTERNAL" An external program running the SCIL interpreter
NUMBER	Process number as an integer.  For the MAIN process and EXTERNAL processes, the value is 0.  For a PICO process, this is the monitor number  For a REPR process, the number is coded as 100 * apl + n, where 'apl' is the application number 'n' = 1 for the time channel queue, 'n' = 2 for the event channel queue and 'n' = 2 + p for parallel queue 'p'.  For a PRIN process, the number is coded as 100 * apl + n, where 'apl' is the application number 'n' = 1 for process printouts and 'n' = 2 for report printouts.  For a PRNC process, this is the printer number.

This function can be used, for example, in the start program of a picture to determine whether the picture is being displayed on a monitor or printed.

#### Example:

```
#LOCAL HOST, OX, CX, OV, S
HOST = SCIL_HOST
#IF HOST.NAME == "PRIN" #THEN #BLOCK
    OX = 'LN':POX'IX'
    CX = 'LN':PCX'IX'
    OV = 'LN':POV'IX'
    S = PRINT_TRANSPARENT((-2,TIMES, "OBJECT TEXT:",OX,-
                           "COMMENT TEXT:",CX, " OBJECT VALUE:",-1,DEC(OV)))
#BLOCK_END
```

This program block first checks whether the picture is being printed or shown on the screen. If it is printed, the PRINT\_TRANSPARENT function prints a row containing the present time and the values of attributes OX, CX and OV of a process object.

'apl' is the application number  
 'n' = 1 for the time channel queue,  
 'n' = 2 for the event channel queue and  
 'n' = 2 + p for parallel queue 'p'.  
 'apl' is the application number  
 'n' = 1 for process printouts and  
 'n' = 2 for report printouts.

## 10.11 Functions related to the programming environment

### 10.11.1 COMPILE(source)

Runs the SCIL compiler.

'source'	Text vector containing the SCIL source code.
Value:	A list value with following attributes:
STATUS	Integer, status code from the compilation
CODE	Byte string, the compiled byte code
ERROR_LINE	Text, the erroneous source line
ERROR_LINE_NUMBER	Integer, the line number in error
ERROR_POSITION	Integer, the character position in error

Attribute CODE is returned if the compilation succeeds (STATUS == 0). Attributes ERROR\_LINE, ERROR\_LINE\_NUMBER and ERROR\_POSITION are returned if the compilation fails.

### 10.11.2 MAX\_APPLICATION\_NUMBER

Maximum number of application objects.

Value:	Integer, 250 (99 in rev. 8.4.4)
--------	---------------------------------

### 10.11.3 MAX\_BIT\_STRING\_LENGTH

Maximum number of bits in a bit string type value.

Value:	Integer, 65535
--------	----------------

### 10.11.4 MAX\_BYTE\_STRING\_LENGTH

Maximum number of bytes in a byte string type value.

Value:	Integer, 8 388 600 (1 048 576 in rev. 8.4.4)
--------	--

### 10.11.5 MAX\_INTEGER

Largest positive integer value.

Value:	Integer, 2 147 483 647
--------	------------------------

### 10.11.6 MAX\_LINK\_NUMBER

Maximum number of link objects.

Value:	Integer, 20
--------	-------------

**10.11.7 MAX\_LIST\_ATTRIBUTE\_COUNT**

Maximum number of attributes in a list.

Value: Integer, 2 000 000 (10 000 in rev. 8.4.4, 1 000 000 in rev. 8.4.5 - 9.4)

**10.11.8 MAX\_MONITOR\_NUMBER**

Maximum number of monitor objects.

Value: Integer, 100 (50 in MicroSCADA rev. 8.4.5).

**10.11.9 MAX\_NODE\_NUMBER**

Maximum number of node objects.

Value: Integer, 250

**10.11.10 MAX\_OBJECT\_NAME\_LENGTH**

Maximum length of application and Visual SCIL object names.

Value: Integer, 63

**10.11.11 MAX\_PICTURE\_NAME\_LENGTH**

Maximum length of picture names.

Value: Integer, 10

**10.11.12 MAX\_PRINTER\_NUMBER**

Maximum number of printer objects.

Value: Integer, 20

**10.11.13 MAX\_PROCESS\_OBJECT\_INDEX**

Maximum number of process objects in a process object group.

Value: Integer, 65 535 (10 000 in rev. 8.4.4)

**10.11.14 MAX\_REPRESENTATION\_NAME\_LENGTH**

Maximum length of representation names.

Value: Integer, 10

## 10.11.15 MAX\_STATION\_NUMBER

Maximum number of station objects.

Value: Integer, 50 000 (in 9.2 SP1, 5 000 in 9.2)

## 10.11.16 MAX\_STATION\_TYPE\_NUMBER

Maximum number of station type objects.

Value: Integer, 33 (31 in rev. 9.1)

## 10.11.17 MAX\_TEXT\_LENGTH

Maximum number of characters in a text type value.

Value: Integer, 65 535 (255 in rev. 8.4.4)

## 10.11.18 MAX\_VECTOR\_LENGTH

Maximum number of elements in a vector.

Value: Integer, 2 000 000 (10 000 in rev. 8.4.4, 1 000 000 in rev. 8.4.5 - 9.4)

## 10.11.19 MAX\_WINDOW\_NAME\_LENGTH

Maximum length of window and picture function names.

Value: Integer, 10

## 10.11.20 MIN\_INTEGER

Smallest negative integer value.

Value: Integer, -2 147 483 648

## 10.11.21 OBJECT\_ATTRIBUTE\_INFO(apl, type [,subtype [,selection]])

Returns the properties of application or system object attributes.

'apl'	Integer value, the logical application number (0 = current). Both local and external applications are supported.
'type'	Text keyword value specifying the object type:
"P"	Process object
"X"	Scale object
"H"	Event handling object
"F"	Free type object
"D"	Data object

Table continues on next page

	"C"	Command procedure object
	"T"	Time channel object
	"A"	Event channel object
	"G"	Logging profile objects
	"B"	Base system object
	"OBJECT"	Attribute info is requested from the named object given as the next argument.
'subtype'		If 'type' = "OBJECT", text value specifying the object name. Otherwise, its value depends on the 'type':
	'type' = "P":	Integer 0 or keyword value "COMMON". The properties of the common attributes of process objects are returned. Integer >0, taken as the process object type PT. The properties of the PT specific attributes are returned.
		Text value, either the two-letter mnemonic name of a predefined process object type or the name of the F-object describing the type. The properties of the PT specific attributes are returned.
		If the 'subtype' argument is omitted, the properties of the attributes of the process group are returned.
	'type' = "B":	Text value, the three-letter mnemonic name of the base system type.
	'type' = "H":	Text value "SYS" or "AEC", the three-letter value of the HT (Event Handling Type) attribute. If empty or omitted, all attributes are considered.
	'type' = "G":	Text value "OBJECT", "DATABASE" or "HISTORY", the value of the PT (Profile Type) attribute.
	Other types:	Omitted or an empty text.
'selection'		Text value that selects the attributes whose properties are returned:
	"ALL"	All attributes. This is the default value.
	"CONFIGURATION"	Configuration attributes. For application objects, these are the attributes returned by the FETCH function.
	"DYNAMIC"	Dynamic attributes.
	aa	The two-letter name of the attribute, whose properties are returned.
Value:		A list value with the following attributes, if 'selection' = aa, otherwise a vector of such list values:
	SHORT_NAME	The two-letter name of the attribute
	LONG_NAME	The two-word name of the attribute
	DESCRIPTION	The text identifier of the description of the attribute.
	ACCESS	Text vector containing one or more of the following keywords: "READ", "WRITE", "SET", "MODIFY", "SUBSCRIBE". See below.
	VALUE_TYPE	The data type of the attribute. See below.
	DEFAULT	The default value of the object attribute. Omitted if the attribute is not a configuration attribute or the attribute has no default value.

Table continues on next page

VECTOR_LENGTH	The maximum length of the vector attribute. Omitted if not a vector.
ELEMENT_TYPE	The data type of the elements of the vector attribute. Omitted if not a vector. See below.
ELEMENT_DEFAULT	The default value of the elements of the vector attribute. Omitted if not a vector or the attribute is not a configuration attribute or the elements have no default value.

The ACCESS of the attribute is defined as follows:

"READ"	The attribute is readable.
"WRITE"	The attribute may be written by the #SET command and by the OPC Data Access Server.
"SET"	The (process object) attribute may be written by the so-called list set command (for example, #SET ABC:P1 = LIST(OV=1,...) and by the OPC Data Access Server.
"MODIFY"	The attribute may be written by the #MODIFY command and by the OPC Data Access Server.
"SUBSCRIBE"	The attribute may be subscribed to by the OPC Data Access Server using the update rate 0.

The VALUE\_TYPE or the ELEMENT\_TYPE of the attribute is one of the standard SCIL data types or one of the following:

"ANY"	The value type may vary case by case.
"ANALOG"	The value type is either "REAL" or "INTEGER", depending on the configuration of the object.
"DATA"	The value type is "REAL", "INTEGER" or "TEXT", depending on the configuration of the object.

Examples:

```

OBJECT_ATTRIBUTE_INFO(0, "D")
; The properties of all attributes of data objects
OBJECT_ATTRIBUTE_INFO(0, "D", "", "CONFIGURATION")
;The properties of the configuration attributes of data objects
OBJECT_ATTRIBUTE_INFO(0, "D", "", "LF")
;The properties of the LF attribute of data objects
OBJECT_ATTRIBUTE_INFO(0, "P", "COMMON")
;The properties of the common attributes of process objects
OBJECT_ATTRIBUTE_INFO(0, "P", "BI")
;The process object type specific attributes of binary input objects
OBJECT_ATTRIBUTE_INFO(2, "P", "FREETYPE")
;The process object type specific attributes of objects of type
FREETYPE
OBJECT_ATTRIBUTE_INFO(0, "OBJECT", "ABC:P1")
;The properties of all attributes of the object ABC:P1
OBJECT_ATTRIBUTE_INFO(0, "B", "STA")
;The properties of the attributes of the STA base system objects

```

## 10.11.22 STATUS\_CODE(mnemonic)

The numeric value of a mnemonic status code name.

'mnemonic'	Text keyword value, the mnemonic status code name.
Value:	Integer value, the numeric value of the status code. -1, if there is no status code by given name.

See function STATUS\_CODE\_NAME for the reverse operation and examples.

### 10.11.23 STATUS\_CODE\_NAME(code)

The mnemonic name of a numeric status code.

'code'	Integer, the numeric status code
Value:	Text value, the mnemonic name of the status code. Empty string, if status code number is not used.

Examples:

The following statements are true:

```
STATUS_CODE ("SCIL_UNDEFINED_VARIABLE") == 188
STATUS_CODE_NAME(188) == "SCIL_UNDEFINED_VARIABLE"
STATUS_CODE ("NO SUCH_ERROR_CODE") == -1
STATUS_CODE_NAME(-1) == ""
```

### 10.11.24 VALIDATE(as, string)

Validates a text string as a SCIL object name.

'as'	Text keyword value telling how to interpret 'string': "VS_OBJECT_NAME", "APPLICATION_OBJECT_NAME", "WINDOW_NAME", "PICTURE_NAME" or "VARIABLE_NAME".
'string'	Text string to be validated.
Value:	Integer, SCIL status code. 0 = OK.

This function validates a given text string as a VS object, an application object, a window, a picture or a variable name. If the name is OK, integer zero is returned. Otherwise an appropriate SCIL status code is returned. Only the validity of the name is checked, existence of the object is not verified.

Example:

```
#LOCAL STATUS
#LOCAL INPUT_STRING = "A_B_C"
STATUS = VALIDATE("VS_OBJECT_NAME", INPUT_STRING)
#IF STATUS == 0 #THEN ROOT.CREATE_NEW_OBJECT(INPUT_STRING)
#ELSE MESSAGES.SHOW("INVALID OBJECT NAME: " + STATUS_CODE_NAME(STATUS))
```

### 10.11.25 VALIDATE\_OBJECT\_ADDRESS(apl, pt, un, oa [,subaddress] [,self])

Validates an object address.

The function returns the status code the base system would return, if an object by the given object type and address would be created. In case of an address overlap, it reports the conflicting process object.

'apl'	Integer value, the logical application number (0 = current). Both local and external applications are supported.
'pt'	Integer value, the process object type: 3 BI 5 BO 6 DI 7 DO 9 AI 11 AO 12 DB 13 PC 14 BS 15 FT 16 OE 17 NT
'un'	Integer value, the unit number.
'oa'	The object address. Either an integer value, the OA attribute of the object, or a text value, the IN attribute of an OPC object
'subaddress'	The object subaddress. Either an integer value, the object bit address (OB attribute), or a text keyword "IN" or "OUT" for OPC objects. "IN" stands for the input object by the item name 'oa', "OUT" for the output object by the item name 'oa'. Default value 16 (no bit address) for numeric addresses, "IN" for textual (OPC) addresses.
'self'	A list with the following attributes: LN Text, the logical name of the process object being validated IX Integer, the logical index of the process object being validated  This attribute may be specified when the object address of an existing process object is changed. Address overlap is not reported if the conflicting process object is 'self'.
Value:	A list with the following attributes: STATUS Integer status code, the result of validation: 0 The address is valid, no conflict. 2129 PROF_PT_ATTRIBUTE_OUT_OF_RANGE 2131 PROF_BIT_ADDRESS_MISSING 2134 PROF_BIT_ADDRESS_NOT_ALLOWED 2136 PROF_PHYSICAL_ADDRESS_OVERLAP See the attributes LN and IX below. 2314 PROF_OBJECT_ADDRESS_NOT_ALLOWED  The following two attributes are returned only when STATUS = 2136: LN Text, the logical name of the overlapping object. IX Integer, the index of the overlapping object.

## 10.12 Language functions

This section describes the SCIL functions that are related to translating different application texts into different languages.

The translations of the texts included in the Visual SCIL objects, such as button labels, menu texts etc., are normally defined by using the Dialog Editor and stored in the same objects. The function TRANSLATE is used to translate the texts into the operator's language.

The translations of other application texts, such as texts describing process objects and their states, are stored in text databases. These texts are translated explicitly by the TRANSLATION function, or usually implicitly by referencing a language sensitive attribute of the object.

## 10.12.1 Language identifiers

It is recommended, but not required, that the two-letter language identifiers defined by the ISO standard 639 are used by the applications.

When ISO 639 language identifiers are used, the system is able to map the Windows language id's, which are derived from the Windows locale id's, to the language identifiers of the SYS600 applications. Consequently, the OPC clients connected to the OPC Data Access Server may define their language by the means specified in the OPC standard, and the base system automatically converts the Windows language id's to the ISO 639 language identifiers.

The applications should select the language identifiers from [Table 5](#).

*Table 5: ISO 639 language identifiers and Windows language id's*

Afrikaans	AF	LANG_AFRIKAANS
Albanian	SQ	LANG_ALBANIAN
Arabic	AR	LANG_ARABIC
Armenian	HY	LANG_ARMENIAN
Assamese	AS	LANG_ASSAMESE
Azerbaijani	AZ	LANG_AZERI
Basque	EU	LANG_BASQUE
Byelorussian	BE	LANG_BELARUSIAN
Bengali	BN	LANG_BENGALI
Bulgarian	BG	LANG_BULGARIAN
Catalan	CA	LANG_CATALAN
Chinese	ZH	LANG_CHINESE
Croatian	HR	LANG_CROATIAN
Czech	CS	LANG_CZECH
Danish	DA	LANG_DANISH
Dutch	NL	LANG_DUTCH
English	EN	LANG_ENGLISH
Estonian	ET	LANG_ESTONIAN
Faroese	FO	LANG_FAEROESE
Persian	FA	LANG_FARSI
Finnish	FI	LANG_FINNISH
French	FR	LANG_FRENCH
Georgian	KA	LANG_GEORGIAN
German	DE	LANG_GERMAN
Greek	EL	LANG_GREEK
Table continues on next page		

Gujarati	GU	LANG_GUJARATI
Hebrew	HE	LANG_HEBREW
Hindi	HI	LANG_HINDI
Hungarian	HU	LANG_HUNGARIAN
Icelandic	IS	LANG_ICELANDIC
Indonesian	ID	LANG_INDONESIAN
Italian	IT	LANG_ITALIAN
Japanese	JA	LANG_JAPANESE
Kannada	KN	LANG_KANNADA
Kashmiri	KS	LANG_KASHMIRI
Kazakh	KK	LANG_KAZAK
Korean	KO	LANG_KOREAN
Latvian	LV	LANG_LATVIAN
Lithuanian	LT	LANG_LITHUANIAN
Macedonian	MK	LANG_MACEDONIAN
Malay	MS	LANG_MALAY
Malayalam	ML	LANG_MALAYALAM
Marathi	MR	LANG_MARATHI
Nepali	NE	LANG_NEPALI
Norwegian	NO	LANG_NORWEGIAN
Oriya	OR	LANG_ORIYA
Polish	PL	LANG_POLISH
Portuguese	PT	LANG_PORTUGUESE
Punjabi	PA	LANG_PUNJABI
Romanian	RO	LANG_ROMANIAN
Russian	RU	LANG_RUSSIAN
Sanskrit	SA	LANG_SANSKRIT
Serbian	SR	LANG_SERBIAN
Sindhi	SD	LANG_SINDHI
Slovak	SK	LANG_SLOVAK
Slovenian	SL	LANG_SLOVENIAN
Spanish	ES	LANG_SPANISH
Swahili	SW	LANG_SWAHILI
Swedish	SV	LANG_SWEDISH
Tamil	TA	LANG_TAMIL
Tatar	TT	LANG_TATAR
Telugu	TE	LANG_TELUGU
Thai	TH	LANG_THAI
Turkish	TR	LANG_TURKISH
Ukrainian	UK	LANG_UKRAINIAN
Urdu	UR	LANG_URDU
Uzbek	UZ	LANG_UZBEK
Table continues on next page		

Vietnamese	VI	LANG_VIETNAMESE
Uzbek	UZ	LANG_UZBEK
Vietnamese	VI	LANG_VIETNAMESE

## 10.12.2 The language of the SCIL context

When a SCIL context is created, it is assigned to an initial language according to the following rules:

1. If the context is owned by a monitor, i.e. it is a SCIL context of a picture or a Visual SCIL object, the LA (Language) attribute of the monitor (MON:BLA) defines the initial language.
2. If the context is not owned by a monitor, the LA attribute of the application (APL:BLA) defines the initial language of the SCIL context.

The language may later be changed by the SET\_LANGUAGE function. This function may also be used to restore the initial language of the SCIL context.

When SYS600 is accessed via the OPC Data Access Server, the language is chosen by the functions specified in the OPC standard. See the System Objects manual for the descriptions of the LA attributes.

The language of the SCIL context has the following meanings:

- When a language sensitive attribute of an application object is read, it is automatically translated into the language of the SCIL context. Examples of such attributes are the TX (Translated Object Text) and the SX (Translated Object State) attribute of a process object.
- The language of the SCIL context works as the second argument's default value of the SCIL TRANSLATE and TRANSLATION functions, see below.



When a language sensitive attribute of an external application is evaluated (by using the APL-APL communication), the attribute is translated into the language of the external application, by default. If an explicit translation into another language is wanted, the language of the SCIL context must explicitly be set by the SET\_LANGUAGE function before the evaluation of the attribute.

## 10.12.3 Text databases

The translations of application texts are stored in data files called text databases.

The databases have three different scopes for different needs of software components using translated texts:

1. The application text database APL\_TEXT.SDB is designed to contain the site-specific texts of the application.
2. The text databases listed by the application attribute APL:BTD are to be used by various software products, such as LIBxxx products, and their localizations.
3. The system text database SYS\_TEXT.SDB is delivered with the SYS600 base software and should not be modified.

The databases are searched in the scope order, APL\_TEXT.SDB first and SYS\_TEXT.SDB last.

The text databases are organized as SCIL databases, where the text identifier acts as the section name. The databases are maintained by the Text Translation Tool, or directly by using the SCIL function DATA\_MANAGER, see the example below.

Example:

The following piece of SCIL code creates a translation of the text identifier "IDClosed" into a couple of languages:

```
#local db,-
    ok = 0,-
    result,-
    id,-
    translations,-
    close_result

id = "IDClosed"
translations = list(EN = "Closed", DE = "Geschlossen", FI = "Kiinni")

db = data_manager("OPEN", "APL_TEXT.SDB")
#if db.status <> ok #return db.status

result = data_manager("CREATE_SECTION", db.handle, id)
#if result.status == ok #then -
    result = data_manager("PUT", db.handle, id, translations)
close_result = data_manager("CLOSE", db.handle)
#return result.status
```

## 10.12.4 SET\_LANGUAGE(*language*)

Sets the current language of the SCIL context.

'language'	A text keyword, language identifier as defined in the ISO standard 639.
Value:	A text keyword, the language of the SCIL context before the function call.

The function does not check that the language identifier is a valid ISO 639 language identifier, nor does it check that the language is really supported by the application.

If the 'language' argument is an empty string, the initial language of the SCIL context is restored.

## 10.12.5 TRANSLATE(*text* [,*language*])

Translates texts defined in Visual SCIL objects.

'text'	A text value, a reference to a text identifier defined in the Dialog Editor, or a vector of such text identifiers. The reference starts with an @ character.
'language'	A text keyword, language identifier as defined in the ISO standard 639.
Value:	A text or a text vector value, the translated text(s).
	If there is no initial @ character in 'text', no translation is done, but the text is returned as such. If there are two initial @ characters, one of the @ characters is removed from the resulting text, but no translation takes place.

The function can only be used in user interface objects (Visual SCIL objects and pictures). The function searches the current dialog or dialog item (object THIS) for the text reference. If either the text reference or the language is not found, the parent object is searched, and so on, up to the object that was loaded with .LOAD command.

The 'language' argument is used only if the language selected for the monitor should be overridden. If no 'language' argument is given, the language of the SCIL context is used (see above).

## 10.12.6 TRANSLATION(id [,language])

Translates texts by using text databases.

'id'	A text value, the identifier of the text to be translated, or a vector of such identifiers.  The identifier is case-sensitive, may be of any length and may contain any characters.
'language'	A text keyword, language identifier as defined in the ISO standard 639.
'Value:'	A text or a text vector value, the translated text(s).

Normally, the function is used without the 'language' argument. In this case, the language of the SCIL context is used (see above). If no translation into the language is found, the English translation is returned, if any.

If the target language is explicitly given as the argument 'language' and the translation is not found, no automatic translation into English is performed.

If no translation into the requested language is found, the function returns the text identifier as such.

### Example

```
.SET OBJECT_TEXT._TITLE = TRANSLATION('LN':POX'IX')
; Display the OX attribute in the operator's native language
; Because the TX attribute is the translation of OX, this is equivalent to
.SET OBJECT_TEXT._TITLE = 'LN':PTX'IX'
```

## 10.13 Error tracing functions

The functions in this section are used for debugging SCIL programs. Tracing means recording of all SCIL statements that are executed. The tracing is started by the TRACE\_BEGIN function and stopped by TRACE\_END. If TRACE\_BEGIN is called while tracing is already on, the second call is ignored but counted: two TRACE\_ENDs are needed to stop tracing. Functions TRACE\_PAUSE and TRACE\_RESUME are used to skip tracing of uninteresting parts of the program execution.

The statements that contain variable expansions (macros), are recorded in their expanded form. Each line is preceded by the depth of the control structure hierarchy to help find matching BLOCKs and BLOCK\_ENDs etc. Optionally, a time tag is inserted at the beginning of each line.

The function SCIL\_LINE\_NUMBER helps to generate programmed tracing information.

### 10.13.1 SCIL\_LINE\_NUMBER

Tells the current line number within the SCIL program.

Value:	Integer value. The current line number
--------	--

### Example:

```
@C = CONSOLE_OUTPUT("Reached line " + DEC(SCIL_LINE_NUMBER))
```

## 10.13.2 TRACE\_BEGIN(filename [,append] [,time\_tags] [,no\_cyclics])

Starts trace logging.

'filename'	Text or byte string (file tag). The name of the file where the trace output is written. See <a href="#">Section 6.5.1</a> for file naming.
'append'	Text keyword "APPEND". If given, the trace output is appended to the file if it already exists.
'time_tags'	Text keyword "TIME_TAGS". If omitted, no time tags are written.
'no_cyclics'	Text keyword "NO_CYCLICS". If given, cyclic methods of Visual SCIL objects and update programs of pictures are not traced.
Value:	Integer value. The status of file creation (0 = successful).

The keyword arguments 'append', 'time\_tags' and 'no\_cyclics' may be given in any order.

## 10.13.3 TRACE\_END

Stops trace logging.

Value:	Integer value. The status of closing the trace file (0 = successful).
--------	---

## 10.13.4 TRACE\_PAUSE

Pauses trace logging.

Value:	Integer value. The status the operation (0 = successful).
--------	---

## 10.13.5 TRACE\_RESUME

Resumes trace logging.

Value:	Integer value. The status of the operation (0 = successful).
--------	--

## 10.14 Database functions

### 10.14.1 General object listing functions

### 10.14.2 APPLICATION\_OBJECT\_ATTRIBUTES(apl, type, objects, attributes)

Reads the values of specified attributes of given application objects.

'apl'	Logical application number, see APPLICATION_OBJECT_LIST.
'type'	Object type, see APPLICATION_OBJECT_LIST.
'objects'	A text vector containing the names of the objects or a list with a text vector attribute LN containing the names. In the case where the 'type' argument (second argument) is "IX" or "IX_AND_UP", the 'objects' argument is a list with two attributes: LN Text vector containing the object names

Table continues on next page

	IX	Integer vector containing the indices
'attributes'		Text vector containing the names of attributes to be read. The vector may contain vendor attributes, i.e. attributes named by the caller and defined by SDDL (SCIL Data Derivation Language). See example below.
Value:		A list with attributes specified by 'attributes'. Each attribute is a vector containing the values of that attribute in the specified objects.

**Example:**

```

@OBJECTS = LIST(LN = ("BREAKER", "BREAKER"), IX = (10, 22))
@ATTRS = ("OV", "OS", "RT_STR = TIMEMS(RQ)") ;The last one is a
                                                ;named RT_STR evaluated
                                                ;by SDDL
                                                ;expression TIMEMS(RQ)
@VALUES = APPLICATION_OBJECT_ATTRIBUTES(0, "IX", %OBJECTS, %ATTRS)
@OV10 = %VALUES.OV(1)
@OS10 = %VALUES.OS(1)
@RT_STR10 = %VALUES.RT_STR(1)
@OV22 = %VALUES.OV(2)
@OS22 = %VALUES.OS(2)
@RT_STR22 = %VALUES.RT_STR(2)

```

The value, status and registration time of two indices of process object BREAKER1 are read.  
Note that this code is not equivalent to the following:

```

@OV10 = BREAKER:POV10
@OS10 = BREAKER:POS10
@RT_STR10 = TIMEMS(BREAKER:PRQ10)
@OV22 = BREAKER:POV22
@OS22 = BREAKER:POS22
@RT_STR22 = TIMEMS(BREAKER:PRQ22)

```

Use of APPLICATION\_OBJECT\_ATTRIBUTES guarantees that the attribute values are from the same moment of time. In the latter example, the database may have been updated between the first and last command.

### 10.14.3 APPLICATION\_OBJECT\_COUNT(apl, type [,order [,direction [,start [,condition]]]])

Counts application objects that fulfil given conditions.

'apl'	Logical application number, see APPLICATION_OBJECT_LIST.
'type'	Object type, see APPLICATION_OBJECT_LIST.
'order'	Search order, see APPLICATION_OBJECT_LIST.
'direction'	Search direction, see APPLICATION_OBJECT_LIST.
'start'	Start point, see APPLICATION_OBJECT_LIST.
'condition'	Search condition, see APPLICATION_OBJECT_LIST. The condition is evaluated in read-only mode.
Value:	Integer. The number of objects of the given type fulfilling the condition.



Because of modifications to types "P" and "IX", the behaviour in revision 8.4.2 is not fully compatible with 8.4.1. Types "P" and "IX" no longer count process objects of user defined types.

**Example:**

```
#LOCAL ALARMS_IN_UNIT_5, TIME_CHANNEL_COUNT
ALARMS_IN_UNIT_5 = APPLICATION_OBJECT_COUNT(0, "IX", "UNIT", "", 5, "AL
== 1")
TIME_CHANNEL_COUNT = APPLICATION_OBJECT_COUNT(0, "T")
```

## 10.14.4 APPLICATION\_OBJECT\_EXISTS(apl, type, name [,condition [,verbosity]]])

Checks whether an application object exists.

'apl'	Logical application number, see APPLICATION_OBJECT_LIST.
'type'	Object type, see APPLICATION_OBJECT_LIST.
'name'	The name of the object being searched for. In case of a process object ('type' = "IX"), a list with attributes:  LN                         Text, the logical name of the object IX                         Integer 1 ... 65535, the index of the object  For other object types, a text containing the name of the object.
'condition'	Additional optional criterion to be fulfilled to consider the object as existing. For details, see APPLICATION_OBJECT_LIST. The condition is evaluated in read-only mode.
'verbosity'	Optional text keyword, which specifies the form of the function result:  "LACONIC" or ""             Boolean result. This is the default value. "VERBOSE"                     List result.
Value:	When verbosity is "LACONIC", a Boolean value:  TRUE                         The object exists. FALSE                         The object does not exist or does not fulfil the given condition.  When verbosity is "VERBOSE", a list of following attributes:  STATUS                         SCIL status code, 0 = OK_STATUS. The following attribute is present only if STATUS = 0. EXISTS                         Boolean value, as above.

Example:

```
; Do something if ABC:P1 exists and is connected to the process
#IF APPLICATION_OBJECT_EXISTS(0, "IX", LIST(LN="ABC", IX=1), -
"IU == 1 AND SS == 2") #THEN .DO_SOMETHING
```

## 10.14.5 APPLICATION\_OBJECT\_LIST(apl, type [,order [,direction [,start [,condition [,attributes [,max]]]]]])

Lists application objects that fulfil given conditions.

'apl'	Integer. The logical number of the application to navigate in. 0 = current application. Both local and external applications are supported.
'type'	Text keyword. Object type: "P", "IX", "UP", "IX_AND_UP", "H", "X", "F", "D", "C", "T", "A" or "G".  "P"                         Process object groups. "IX"                         Indices of process objects of predefined types. "UP"                         Process objects of user defined types.

Table continues on next page

	"IX_AND_UP"	Process objects of both predefined and user defined types.
	"H"	Event handling objects
	"X"	Scales.
	"F"	Free type objects.
	"D"	Data objects.
	"C"	Command procedures.
	"T"	Time channels.
	"A"	Event channels.
	"G"	Logging profiles.
'order'	Text keyword. The search order given as a text expression:	
	"A" or "ALPHABETIC"	Alphabetical order. When used with type "P", only group names are included in the search. If used with type "IX" or "IX_AND_UP", the indices are searched in name/index order.
	"I" or "INDEX"	Index order (only for process objects of predefined types). Searches the indices of the process object group given as 'start'.
	"P" or "PHYSICAL"	Address order. Applicable only when 'type' is "IX", "UP" or "IX_AND_UP".
	"U" or "UNIT"	Alphabetic (name/index) order within the unit given as 'start' (types "IX", "UP" and "IX_AND_UP" only).
	"OI"	Object identifier order within the object identifier(s) specified by 'start' (types "IX", "UP" and "IX_AND_UP" only). The objects with the same OI are ordered by LN and IX.
	"E" or "EXECUTION"	Execution order within a time channel. 'type' can be either "D" or "C". Whichever is given, both data objects and command procedures are searched.
	""	Default value (= "A")
'direction'	Text keyword:	
	"F" or "FORWARD"	Forward browsing
	"B" or "BACKWARD"	Backward browsing
	""	Default value (= "F")
'start'	Start point of the search. Depends on the object type and search order as follows::	
	Order "A":	Type "IX" and "IX_AND_UP": Logical name (text), or a list with the following attributes:
	LN	Logical name (text)
	IX	Index (integer)
	Other types	Logical name (text)
	All types:	"" = default value
	Order "I":	Logical name (text)
	Order "P":	Unit (integer), or a list with the following attributes:
	UN	Unit (integer)
	OA	Object address (integer) (optional) Stations with numeric addresses only
	OB	Object bit address (integer) (optional) Stations with numeric addresses only
	IN	Item name (text) (optional) OPC and OAE stations only

Table continues on next page

	IN_OUT	Text keyword "IN" or "OUT" (optional) OPC stations only, default = "IN"
	"" = default value	
Order "U":	Unit number (integer), or a list with the following attributes:	
	UN	Unit (integer)
	LN	Logical name (text) (optional)
	IX	Index (integer) (optional)
Order "OI":	Object identifier (text). If the last character of the argument is *, the process objects whose OI begins with the argument string are listed.	
Order "E":	Time channel name (text), or a list with the following attributes:	
	TC	Time channel name (text)
	OT	Object type: "D" or "C" (optional)
	ON	Object name (text) (optional)
		If the 'start' argument specifies an existing object, that object is not included in the search. The 'start' argument is case insensitive.
'condition'	A text containing the criterion for selecting objects.	
		The selection criterion is a boolean type expression composed of relations and logical operators. The relations have an attribute as the left operand. All attributes, except vector and list type attributes, can be included in the expression. In conjunction with text attributes, the wildcard characters % and * can be used. % matches any character, * matches any sequence of characters (including a zero length sequence).
		Default = "".
		The condition is evaluated in read-only mode.
'attributes'	A text or text vector containing the name(s) of attribute(s) to be returned in addition to LN (and IX or OT). The vector may contain vendor attributes, i.e. attributes named by the caller and defined by SDDL (SCIL Data Derivation Language). See the example below. Default = "" (no additional attributes).	
'max'	Maximum number of objects to be returned (integer). Default = 10 000.	
Value:	A list including the following attributes:	
	COUNT	Integer value, number of objects returned
	MORE	Boolean value. TRUE if browsing was interrupted due to 'max' being exceeded
	LN	Text vector, names of the objects
	IX	Integer vector, indices (types "IX" and "IX_AND_UP" only)
	OT	Text vector (values "D" or "C") (order "E" only)
		Plus additional attributes defined by the 'attributes' argument.
		The objects are returned in the order specified by 'order' even if backward browsing is specified.



Because of modifications to types "P" and "IX", the behaviour in revision 8.4.2 is not fully compatible with 8.4.1. Types "P" and "IX" no longer return process objects of user defined types.

#### Example 1:

```
APPLICATION_OBJECT_LIST(0, "IX")
; All (or 10000 first) process objects (of predefined type)
APPLICATION_OBJECT_LIST(0, "IX", "UNIT", "", 5)
; All objects of unit 5
APPLICATION_OBJECT_LIST(0, "IX", "UNIT", "", 5, "AL == 1")
; Alarming objects of unit 5
APPLICATION_OBJECT_LIST(0, "IX", "UNIT", "", 5, "AL == 1", -
("OV", "OS", "RT_STR = TIMEMS(RQ)"))
; SDDL used here
```

```

; Alarming objects of unit 5,
; attributes LN, IX, OV, OS and RQ are read

    APPLICATION_OBJECT_LIST(0, "IX", "OI", "", "Eastwick", "HB == 1")
; Event blockings in the substation level of Eastwick (OI = "Eastwick")
APPLICATION_OBJECT_LIST(0, "IX", "OI", "", "Eastwick *", "HB == 1")
; Event blockings within the Eastwick substation
APPLICATION_OBJECT_LIST(0, "IX", "OI", "", "Eastwick*", "HB == 1")
; Event blockings within all substations whose name begins with "Eastwick"

    APPLICATION_OBJECT_LIST(2, "T", "ALPHABETIC", "", "", "LN == ""T*L""")
; The time channels in application 2, whose name starts letter T and
; ends with L

```

**Example 2:**

The following piece of code reads and handles all the data objects of the application, 50 objects at a time.

```

#LOCAL OBJECTS
#LOCAL START = ""
#LOCAL MORE_TO_COME = TRUE
#LOOP MORE_TO_COME
    OBJECTS = APPLICATION_OBJECT_LIST(0, "D", "A", "", START, "", "", 50)
    #IF OBJECTS.COUNT > 0 #THEN #BLOCK
        .HANDLE_UP_TO_50_OBJECTS(OBJECTS.LN)
        START = OBJECTS.LN(OBJECTS.COUNT)
    #BLOCK_END
    MORE_TO_COME = OBJECTS.MORE
#LOOP_END

```

## **10.14.6 APPLICATION\_OBJECT\_SELECT(apl, type, names, condition [,verbosity])**

Selects from a list of objects the ones that fulfil the given condition.

'apl'	Logical application number, see APPLICATION_OBJECT_LIST.
'type'	Object type, see APPLICATION_OBJECT_LIST.
'names'	The names of the objects. In case process objects ('type' = "IX" or "IX_AND_UP"), a list with attributes:  LN                          Text vector, the logical names of objects. IX                          Integer vector, the indexes of objects.  For other object types, a text vector containing the names of the objects.
'condition'	The selection criterion. For details, see APPLICATION_OBJECT_LIST. The condition is evaluated in read-only mode.
'verbosity'	Optional text keyword, which specifies the form of the function result:  "LACONIC" or ""              Vector result. This is the default value. "VERBOSE"                    List result.
Value:	When verbosity is "LACONIC", an integer vector containing the selected indexes. For example, if 'names' specify 3 objects and the result is (1, 3), the 1st and 3rd object fulfil the condition, the 2nd does not.  When verbosity is "VERBOSE", a list of following attributes:  STATUS                       SCIL status code, 0 = OK_STATUS. The following two attributes are present only when STATUS = 0. COUNT                        Number of selected objects. SELECTED                    An integer vector containing the selected indexes.

**Example:**

```
; Suppose the value of process objects A:P1 and C:P3 is 0 and
; the value of B:P2 is 1.
;
#LOCAL NAMES = ("A", "B", "C")
#LOCAL INDEXES = (1, 2, 3)
#LOCAL LACONIC = APPLICATION_OBJECT_SELECT(0, "IX", -
    LIST(LN = NAMES, IX = INDEXES), "OV == 0")
#LOCAL VERBOSE = APPLICATION_OBJECT_SELECT(0, "IX", -
    LIST(LN = NAMES, IX = INDEXES), "OV == 0", "VERBOSE")
;Now, the value of variable LACONIC is
;   VECTOR(1, 3) and
;the value of variable VERBOSE is
;   LIST(STATUS = 0, COUNT = 2, SELECTED = VECTOR(1, 3))
;
;To drop the unselected objects from the object list:
NAMES = PICK(NAMES, LACONIC)
INDEXES = PICK(INDEXES, LACONIC)
```

## 10.14.7 **BASE\_SYSTEM\_OBJECT\_LIST(type [,condition [,attributes [,apl]]])**

Lists the base system objects that fulfil the given condition.

'type'	Text keyword. Object type: "SYS", "APL", "MON", "STA", "STY", "PRI", "NOD", "LIN".						
'condition'	A text containing the criterion for selecting objects.  The selection criterion is a boolean type expression composed of relations and logical operators. The relations have an attribute as the left operand. All attributes, except vector and list type attributes, can be included in the expression. In conjunction with text attributes, the wildcard characters % and * can be used. % matches any character, * matches any sequence of characters (including a zero length sequence). Default = "".						
'attributes'	The condition is evaluated in read-only mode.  A text or text vector containing the name(s) of the attributes to be returned in addition to BM (Base System Object Number). The vector may contain vendor attributes, i.e. attributes named by the caller and defined by SDDL (SCIL Data Derivation Language). Default = "" (no additional attributes).						
'apl'	Integer. The logical number of the external application that performs the query. This argument is used to list the base system objects of a remote SYS600 system.						
Value:	<p>A list containing the following attributes:</p> <table border="0"> <tr> <td>STATUS</td> <td>Integer, the SCIL status code of the operation. The following attributes are returned only if STATUS = 0 (OK_STATUS).</td> </tr> <tr> <td>COUNT</td> <td>Integer value, number of objects returned</td> </tr> <tr> <td>BM</td> <td>Integer vector, object numbers of the objects</td> </tr> </table> <p>Plus additional attributes defined by the 'attributes' argument.</p>	STATUS	Integer, the SCIL status code of the operation. The following attributes are returned only if STATUS = 0 (OK_STATUS).	COUNT	Integer value, number of objects returned	BM	Integer vector, object numbers of the objects
STATUS	Integer, the SCIL status code of the operation. The following attributes are returned only if STATUS = 0 (OK_STATUS).						
COUNT	Integer value, number of objects returned						
BM	Integer vector, object numbers of the objects						

The function lists only the objects that do 'exist'. The rules for considering an object as an existing one are the following:

- The SYS object always exists.
- An application (APL) object exists if its NA (Name) is not empty or TT (Translation Type) is not "NONE".
- A station (STA) object exists if its ST (Station Type) is not "NONE" or TT (Translation Type) is not "NONE".
- A station type (STY) object exists if its NA (Name) is not empty.

- A node (NOD) object exists if its LI (Link Number) is non-zero or SA (Station Address) is non-zero or NN (Node Name) is not empty or NT (Node Type) is not "UNKNOWN" or OP (OPC Server Data) is not empty.
- A link (LIN) object exists if its LT (Link Type) is not "NONE".
- For other object types (MON and PRI), an object exists if its DT (Device Type) is not "NONE" or TT (Translation Type) is not "NONE".

**Example:**

```
APLS = BASE_SYSTEM_OBJECT_LIST("APL", "", "NA")
; The numbers and names of the applications in the system

REX_STATIONS = BASE_SYSTEM_OBJECT_LIST("STA", "ST == ""REX""")
; The REX stations in the system

REMOTE_APLS = BASE_SYSTEM_OBJECT_LIST("APL", "", "NA", 10)
; The numbers and names of the applications in a remote system
; i.e. in the system where external application 10 is located
```

## 10.14.8 Object maintenance functions

These functions along with commands #CREATE, #MODIFY, #DELETE and #SEARCH (see [Section 9.2.2](#)) are used to do maintenance of application objects. Function FETCH may also be used to read the attributes of base system objects.

### 10.14.9 **FETCH(apl, type, name [,index])**

Fetches the configuration attributes of an object.

'apl'	Integer expression, 0 ... 250. The logical number of the application. 0 = the own application. The function supports both local and external applications.
'type'	Text expression. The type of the object: "P", "H", "X", "F", "D", "C", "T", "A", "G" or "B".
'name'	Text expression. The name of the object.
'index'	Integer expression, 0 ... 65 535. The index of a process object of a predefined type (not obligatory). If 'index' is omitted or == 0 for process objects of the predefined types, the function returns the attributes that are common to the process object group.
Value:	A list containing the configuration attributes of the object, the dynamic run-time attributes are not returned. If the named object does not exist, a list is returned with only one attribute, IU, which has the value -1.

The function makes a list containing all configuration attributes of any application object (process object, event handling object, scale, free type object, data object, command procedure, time channel, event channel or logging profile).

Additionally, the function may be applied to a base system object. In this case, all the attributes of the base system object are fetched.

**Example:**

```
@V = FETCH(0, "P", "A", 1)
;%V.LN has value "A"
;%V.IX has value 1
```

## 10.14.10 NEXT(n), PREV(n)

Fetches the configuration attributes of an object within a search result.

'n'	Integer, 1 ... 10. The identification number of the search.
Value:	A list containing the configuration attributes of the object, the dynamic runtime attributes are not returned. The object type determines which attributes are returned. If the object does not exist a list is returned containing only the IU attribute which has the value -1.

These functions are used to browse through the result of a search initiated with the #SEARCH command. Process objects, data objects, command procedures, scales, time channels, event channels, logging profiles and free type objects can be searched through. The functions require that the search has been initiated with the #SEARCH command, see [Section 9.2.2](#).

Example:

```
#LOCAL OBJ
#SEARCH 2 0 "P" "A" "A"
OBJ = NEXT(2)
!SHOW NAME OBJ.LN
```

The name of the process object group following A in alphabetical order is shown.

## 10.14.11 PHYS\_FETCH(apl, unit, address [,bit\_address])

Fetches the configuration attributes of a process object

The process object is specified with its physical address.

'apl'	Integer, 0 ... 250. The logical application number. 0 = the own application. Only local applications are supported.
'unit'	Integer. The unit number of the process unit where the object is situated. This is the station number as known to the application.
'address'	Integer. Object address. The OA attribute of the process object.
'bit_address'	Integer, 0 ... 15. Bit number (can be omitted). The OB attribute of the object.
Value:	A list containing the configuration attributes of the object, the dynamic runtime attributes are not returned. If the object does not exist, the list contains only one attribute, IU, which has the value -1.

Example:

```
#LOCAL OBJ = PHYS_FETCH (0,3,1010,5)
```

## 10.14.12 Alarm list functions

### 10.14.13 APPLICATION\_ALARM\_COUNT(apl [, filter])

Counts the alarms and warnings of an application.

'apl'	Integer, 0 ... 250. The logical application number. 0 = the own application. Both local and external applications are supported.
'filter'	A text containing a filter for selecting objects.  The filter is a boolean type expression composed of relations and logical operators. The relations have an attribute as the left operand. All attributes, except vector and list type attributes, can be included in the expression. In conjunction with text attributes, the wildcard characters % and * can be used. % matches any character, * matches any sequence of characters (including a zero length sequence). Default = "". The filter is evaluated in read-only mode.
Value:	A list containing the following integer attributes:
STATUS	SCIL status code, 0 = OK_STATUS. The next attributes are present only if STATUS = 0.
ACTIVE_NOACK	Count of active alarms that do not require an acknowledgement (AL = 1, AR = 1, RC = 0)
ACTIVE_ACKED	Count of acknowledged active alarms (AL = 1, AR = 1, RC = 1)
ACTIVE_UNACKED	Count of unacknowledged active alarms (AL = 1, AR = 0, RC = 1)
FLEETING	Count of fleeting (unacknowledged inactive) alarms (AL = 0, AR = 0, RC = 1)
LOW_WARNINGS	Count of low warnings of analog input objects (AC > 0, AZ = 3)
HIGH_WARNINGS	Count of high warnings of analog input objects (AC > 0, AZ = 4)

If an analog input object is in a warning state and has a fleeting alarm, it is counted as FLEETING until the alarm has been acknowledged.

#### 10.14.14 APPLICATION\_ALARM\_LIST(apl, lists [,attributes [,order [,filter [,max\_count]]]])

Lists the alarms and warnings of an application.

'apl'	Integer, 0 ... 250. The logical application number. 0 = the own application. Both local and external applications are supported.
'lists'	A text or text vector defining the lists to be returned. The following keywords or any combination of them are available:
"ACTIVE_NOACK"	The active alarms that do not require an acknowledgement (AL = 1, AR = 1, RC = 0)
"ACTIVE_ACKED"	The acknowledged active alarms (AL = 1, AR = 1, RC = 1)
"ACTIVE_UNACKED"	The unacknowledged active alarms (AL = 1, AR = 0, RC = 1)
"FLEETING"	The fleeting (unacknowledged inactive) alarms (AL = 0, AR = 0, RC = 1)
"LOW_WARNINGS"	The low warnings of analog input objects (AC > 0, AZ = 3)
"HIGH_WARNINGS"	The high warnings of analog input objects (AC > 0, AZ = 4)

Table continues on next page

	"ACKED"	The acknowledged alarms: "ACTIVE_NOACK" + "ACTIVE_ACKED" (AL = 1, AR = 1)
	"UNACKED"	The unacknowledged alarms: "ACTIVE_UNACKED" + "FLEETING" (AR = 0)
	"ACTIVE"	The active alarms: "ACTIVE_NOACK" + "ACTIVE_ACKED" + "ACTIVE_UNACKED" (AL = 1)
	"ALARMS"	All alarms: "ACTIVE" + "FLEETING"
	"WARNINGS"	All warnings: "LOW_WARNINGS" + "HIGH_WARNINGS"
	"ALL"	All alarms and warnings: "ALARMS" + "WARNINGS"
'attributes'		An optional text or text vector defining the attribute(s) to be returned (apart from LN and IX, which are returned by default). The vector may contain vendor attributes, i.e. attributes named by the caller and defined by SDDL (SCIL Data Derivation Language). See the example below.
'order'		An optional text keyword specifying the order of reported alarms:
	"TIME"	Time order (oldest first). The alarms are ordered according to their AT (and AM) attribute, the warnings according to their WQ attribute.
	"REVERSED" or ""	Reversed time order (newest first) This is the default value.
'filter'		An optional text containing a filter for selecting objects.
		The filter is a boolean type expression composed of relations and logical operators. The relations have an attribute as the left operand. All attributes, except vector and list type attributes, can be included in the expression. In conjunction with text attributes, the wildcard characters % and * can be used. % matches any character, * matches any sequence of characters (including a zero length sequence). Default = "".
		The filter is evaluated in read-only mode.
'max_count'		Integer 1 ... 2000000, the maximum number of alarms to be reported in each list. Optional, the default value is MAX_VECTOR_LENGTH (2000000).
Value:		A list containing the following attributes:
	STATUS	SCIL status code, 0 = OK_STATUS. The next attributes are present only if STATUS = 0.
	ACTIVE_NOACK, ACTIVE_ACKED, ACTIVE_UNACKED, FLEETING, LOW_WARNINGS, HIGH_WARNINGS, ACKED, UNACKED, ACTIVE, ALARMS, WARNINGS, ALL	Any combination of these attributes as defined by the argument 'lists'.
		Each alarm list attribute is a list of following attributes:
	COUNT	The count of alarms and warnings in the list.

Table continues on next page

LN	The logical names of the objects in the list.
IX	The indexes of the objects in the list.
'aa'	The values of each attribute 'aa' defined by the argument 'attributes'.

If an analog input object is in a warning state and has a fleeting alarm, it is listed as FLEETING until the alarm has been acknowledged.

Example:

```
@result = APPLICATION_ALARM_LIST(0, ("ACTIVE", "FLEETING"), -  
("ALARM_TIME = TIMEMS(AQ)", "AC", "AR", "OV"))
```

The result might look like:

STATUS	0	0
ACTIVE	List of	
	COUNT	2
	LN	Vector("ABC", "DEF")
	IX	Vector(1, 3)
	ALARM_TIME	Vector("15-04-21 11:26:28.341", "15-04-21 11:26:14.236")
	AC	Vector(2, 4)
	AR	Vector(0, 1)
	OV	Vector(1, 999.1)
FLEETING	List of	
	COUNT	0

## 10.14.15 Data object functions

Data objects functions read and write the history registrations of data objects.

### 10.14.16 **DATA\_FETCH(apl, name, index1 [,step [,count]]),** **DATA\_FETCH(apl, name, time1, time2 [,step [,shift]]),** **DATA\_FETCH(apl, name, time1 [,step [,count [,shift]]]),** **DATA\_FETCH(apl, name, indices)**

Reads history records of a data object.

'apl'	Integer, 0 ... 250. The logical application number (0 = the own application). External applications are supported.
'name'	Text data. The name of the data object.
'index1'	Integer, 1 ... 2 000 000. The index of the first record to be fetched.
'step'	Integer, positive or negative. Defines the step between records to be fetched. The default value = 1 (all records are fetched). If 'step' is negative, the values are returned in reverse time order.  For example, if 'step' = 2, records 'index1', 'index1' + 2, 'index1' + 4, etc. are fetched.  If 'step' = -2, records 'index1', 'index1' - 2, 'index1' - 4, etc. are fetched.

Table continues on next page

'count'	Integer, 0 ... 2 000 000. The number of records to be fetched. 0 (default) means all existing records up to 10 000.	
'indices'	Vector. An integer vector defining explicitly the records to be fetched.	
'time1'	Time data. The start of the time interval to be fetched.	
'time2'	Time data. The end of the time interval.	
If 'time1' < 'time2', the values are returned in time order regardless of the sign of 'step'. If 'time1' > 'time2', the values are returned in reversed time order regardless of the sign of 'step'. The given time range is regarded as a semi-open range. For example, if the range is given as 09.00.00-10.00.00, a record sampled at 09.00.00 is included but a record sampled at 10.00.00 is excluded (if shift = 0, see below).		
'shift'	Integer, 0 or 1.	
0	No shift	
1	A shift of one sampling interval.	
For example, a record sampled at 10.00.05 is included in the time range 09.00.00-10.00.00, but a record sampled at 09.00.05 is excluded.		
Default value: 0.		
Value:	A list containing the following attributes:	
OV	Real vector, registered data	
RT	Time vector, the registration times	
OS	Integer vector, the status codes	
IX	Integer vector, the indices of the values	
LE	Integer, the number of elements in the result vectors above (OV, RT, OS and IX).	



When a time interval fetch is specified, the function expects that the history records are stored in ascending time order. If not, the set of records included in the result is unspecified.

#### Examples:

```
#LOCAL DATA = DATA_FETCH (0, "ABC", 1, 5, 0)
;Every fifth history record of the data object ABC is read.

#LOCAL T1 = PACK_TIME (1989, 9, 10, 0, 0, 0)
#LOCAL T2 = PACK_TIME (1989, 9, 10, 12, 0, 0)
;Two time data values are defined.

#LOCAL A = DATA_FETCH (0, "ABC", T1, T2)
;The history records of the given time interval are read.

#LOCAL B = DATA_FETCH (0, "DEF", A.IX)
;The corresponding values of the data object DEF.

#LOCAL C = DATA_FETCH (0, "GHI", A.IX +5)
;The corresponding values of GHI shifted by 5.
```

### 10.14.17 DATA\_STORE(apl, name, data, index1 [,step]), DATA\_STORE(apl, name, data, indices)

Writes historical records of a data object.

'apl'	Integer, 0 ... 250. The logical application number. 0 = the own application. External applications are supported.
'name'	Text. The name of the data object.
'data'	List. The values to be stored, 3 attributes: OV      Real vector: recorded values RT      Time vector: registration times OS      Integer vector: status codes
	If the RT attribute is missing from 'data', the original registration times are kept. If the OS attribute is missing, the status codes of the elements of OV vector are stored as the OS attribute.
'index1'	Integer, 1 ... 2 000 000. The index of the first record to be written.
'step'	Integer. This argument defines the step between records to be stored. Default value = 1 (all records are stored). If 'step' is negative, the values are stored in reverse time order.  For example, if 'step' = 2, records 'index1', 'index1' + 2, 'index1' + 4, etc., are stored. If 'step' = -2, records 'index1', 'index1' - 2, 'index1' - 4, etc., are stored.
'indices'	Integer vector. Defines explicitly the records to be written.
Value:	Integer. A status code. See the Status Codes manual. 0 = OK.

**Examples:**

```
#LOCAL HISTORY = DATA_FETCH(0, "A", 1)
#LOCAL STATUS1 = DATA_STORE(0, "DATA", HISTORY, 1)
#LOCAL STATUS2 = DATA_STORE(2, "A", HISTORY, 1)
```

The history of the data object A in the current application is copied to the data object DATA in the current application and to the data object A in application 2.

## 10.14.18 Process object query functions

Process object query functions are used to browse the results of a process object query defined by preceding #INIT\_QUERY command.



These functions are more or less obsolete. Use more powerful SCIL functions APPLICATION\_OBJECT\_LIST, APPLICATION\_OBJECT\_ATTRIBUTES and APPLICATION\_ALARM\_LIST for a query of the process database and HISTORY\_DATABASE\_MANAGER to browse the event history.

## 10.14.19 END\_QUERY

Tells whether a process object query is completed.

Value:	Boolean data. TRUE = the query is completed. FALSE = the query is not completed.
--------	--

The function tests whether all objects matching a process query initiated by the #INIT\_QUERY command ([Section 9](#)) have been read with the PROD\_QUERY function.

**Example:**

```
#IF END_QUERY #THEN #BLOCK
    !SHOW MESSAGE "READY"
    #BLOCK_END
#ELSE #BLOCK
    LIST = PROD_QUERY(20)
```

```

!SHOW NAME LIST.LN
!SHOW VALUE LIST.OV
#BLOCK_END

```

If the query has been completed, the message READY is shown on screen. Otherwise, 20 more process objects are handled.

## 10.14.20 PROD\_QUERY(n)

Returns attributes of objects selected by a process object query.

'n' Integer, [-] 1 ... APL:BQL. The maximum number of process objects that are included in the query. If the number is given with a negative sign, the browsing is performed backwards.

Value: A list containing the following attributes:

Identification: LN (Logical Name), IX (Index), PT (Process Object Type), OI (Object Identification), OX (Object Text)

Object value: OV (Object Value), OS (Object Status)

Alarm state: AL (Alarm), AS (Alarm State), AR (Alarm Receipt), AZ (Alarm Zone)

Time stamps: RT (Registration Time), RM (Registration Milliseconds), AT (Alarm Time), AM (Alarm Milliseconds), YT, YM

RTU attributes: SE (Selection), SP (Stop Execution), OF (Overflow),

Protocol attributes: BL, CT, OR, RA, RB, SB

Blocking: AB, HB, PB, UB, XB

Miscellaneous: RI, RX

In addition, when the query concerns the history buffer: CA (Changed Attribute)

The function returns attribute values of the process objects selected by the preceding #INIT\_QUERY command (see [Section 9](#)). The query concerns always the current application.

After one #INIT\_QUERY, the function may be called several times. Each time it continues from where it finished previously.

The attribute values of a certain attribute form a vector. For example, the LN attribute is a vector of all object names included in the query. Vector elements with the same index refer to the same process object.

Examples:

The result of the function call PROD\_QUERY(4) could be:

Index	Attributes		
	LN	IX	OV
1	"TEMP"	1	63.0
2	"PH"	5	6.5
3	"SWITCH"	1	1
4	"SWITCH"	2	0

The attribute LN (logical name) constitutes the first vector, IX (index) the second one and OV (object value) the third one.

## 10.14.21 History database functions

### 10.14.22 HISTORY\_DATABASE\_MANAGER("OPEN" [,apl])

Opens a session to the history database of an application.

The HISTORY\_DATABASE\_MANAGER function is session-based. A query session is first created. Subsequent calls of the function may then set various query parameters and do queries. Finally, the session is closed.

There may be up to 10 open query sessions within one SCIL context.

When a SCIL context is deleted, the open query sessions are closed automatically. However, it is a good practice to close the sessions explicitly by SCIL to save system resources.

'apl'	Integer or text value. When an integer, specifies the logical application number (0 = current application) Text keyword "NO_APPLICATION" is used to specify no application. Default value is 0.
Value:	List: SESSION Integer value. Used to identify the session in subsequent calls. STATUS Integer value, SCIL status code.

This command opens a new query session and sets the query parameters to their default values (described later). If already 10 sessions are open, status SCIL\_TOO\_MANY\_HDB\_SESSIONS is returned in the STATUS attribute of the result.

If another than the current APPLICATION is specified, it has to be local (in same SYS) but its state is allowed to be COLD.

If no application is specified, SET\_DIRECTORY command must be used to tell the history database manager where to find the database.

If a cold application or no application at all is specified, the user defined attributes of user defined process objects are not returned, because there is no process database to find the attribute descriptions of corresponding F-type objects.

### 10.14.23 HISTORY\_DATABASE\_MANAGER("CLOSE", session)

Closes a session to the history database of an application.

'session'	Integer value returned by OPEN command.
Value:	List: STATUS Integer value, SCIL status code.

This command closes the session by releasing all the resources associated to the session.

### 10.14.24 HISTORY\_DATABASE\_MANAGER("SET\_PERIOD", session, begin [,end])

Sets the time period of the database query.

'session'	Integer value returned by OPEN command.	
'begin'	Time value defining the first day of the period. The following format can be used pack_time(1998,4,1,0,0,0)	
'end'	Time value defining the last day of the period, default = 'begin'.	
Value:	List:  DATE_COUNT                  Integer value. Number of days whose history database was successfully opened.  MISSING_DATES              Time vector containing the dates whose database files could not be read.  MISSING_STATUS             Integer vector containing the status codes of failed database file reads.  STATUS                      Integer value, SCIL status code.	

The period defines the time period whose database files are included in the query. A period may contain up to 1000 database files. Status SCIL\_PERIOD\_TOO\_LONG is returned, if this limit is exceeded.

The arguments 'begin' and 'end' are used only to define the date: hours, minutes and seconds are ignored.

The period is initially empty.

## 10.14.25 HISTORY\_DATABASE\_MANAGER("SET\_DIRECTORY", session, directory)

Sets the location of database files.

'session'	Integer value returned by OPEN command.	
'directory'	Text or text vector containing the directory (or directories) where to locate the database files. See <a href="#">Section 6.5.1</a> for directory naming.	
Value:	List:  STATUS                      Integer value, SCIL status code.	

Up to 20 directories may be specified. Status SCIL\_TOO\_MANY\_DIRECTORIES is returned if this limit is exceeded.

Setting this parameter resets the active period.

When a session is opened, this parameter is set to point to the APL\_subdirectory of the application. If no application is defined, the parameter is left empty.

## 10.14.26 HISTORY\_DATABASE\_MANAGER("SET\_WINDOW", session, begin, end)

Sets the time window of the query.

'session'	Integer value returned by OPEN command.	
'begin'	Time value or an integer 0.  Begin time of the window. If 0, the window has an open beginning.	
'end'	Time value or an integer 0.	

Table continues on next page

Value: List:  
 STATUS Integer value, SCIL status code.

This command sets the time window of the query. A time window may have an open beginning and/or an open end, meaning all events older than 'end' or old events newer than 'begin', respectively.

If both 'begin' and 'end' are non-zero, they may be given in any order.

When a session is opened, the window is set to (0, 0).

## **10.14.27 HISTORY\_DATABASE\_MANAGER("SET\_ORDER", session, order)**

Sets the listing order.

'session' Integer value returned by OPEN command.  
 'order' Text keyword, either "LOG" or "EVENT".  
     "LOG"         The events are returned in the order they were written into the database.  
     "EVENT"        The events are returned in the order specified by the values of attributes ET and EM.  
 Value: List:  
 STATUS          Integer value, SCIL status code.

If logging order is requested, the time window applies to attributes HT and HM of the event, otherwise to attributes ET and EM.

When a session is opened, the order is set to "EVENT".

## **10.14.28 HISTORY\_DATABASE\_MANAGER("SET\_DIRECTION", session, direction)**

Sets the search direction.

'session' Integer value returned by OPEN command.  
 'direction' Text keyword, either "FORWARD" or "BACKWARD".  
     "FORWARD"                   The query starts from the beginning of the time window.  
     "BACKWARD"                  The query starts from the end of the time window.  
 Value: List:  
 STATUS          Integer value, SCIL status code.

The results of a query are returned in the order of 'direction': If "FORWARD", they are returned in time order, if "BACKWARD", they are returned in reversed time order.

When a session is opened, the direction is set to "BACKWARD".

### 10.14.29 HISTORY\_DATABASE\_MANAGER("SET\_TIMEOUT", session, timeout)

Sets the maximum time a query may last.

'session'	Integer value returned by OPEN command.
'timeout'	Integer or real value, time-out in seconds.
Value:	List: STATUS      Integer value, SCIL status code.

The time-out of a query specifies the maximum time a query may last. If it is exceeded, the query is interrupted and the partial results found so far are returned.

When a session is opened, the time-out is set to 5 seconds.

### 10.14.30 HISTORY\_DATABASE\_MANAGER("SET\_CONDITION", session, condition)

Sets the condition for requested events.

'session'	Integer value returned by OPEN command.
'condition'	Text value, the condition requested events should fulfil.
Value:	List: STATUS      Integer value, SCIL status code.

The condition is given as in the APPLICATION\_OBJECT\_LIST function, see above.

When a session is opened, the condition is set to an empty string.

### 10.14.31 HISTORY\_DATABASE\_MANAGER("SET\_ATTRIBUTES", session, attributes)

Sets the attributes whose values are to be returned by the query.

History database information related to each event are described in the Application Objects manual.

'session'	Integer value returned by OPEN command.
'attributes'	Text vector. Specifies the set of attributes whose value is returned by the query.
Value:	List: STATUS      Integer value, SCIL status code.

Any number of attributes may be specified.

Attributes LN, IX, OV, ET and EM do not have to be specified, they are always included in the set. The object type specific names (BI, BO, etc.) for the object value attribute are not used. If no application is specified for the session, the user defined attributes of user defined process objects are not returned by the query.

The initial value of the set is (LN, IX, OV, ET, EM).

The set may contain vendor attributes, i.e. attributes named by the caller and defined by SDDL (SCIL Data Derivation Language). For example, "RT\_STR = TIMEMS(RQ)" is a valid attribute definition, which returns the formatted text representation of the registration time.

## 10.14.32 HISTORY\_DATABASE\_MANAGER("GET\_PARAMETERS", session)

Returns the current values of parameters.

'session'	Integer value returned by OPEN command.	
Value:	List:	
	DIRECTORY	Text vector, file directories to locate database files.
	PERIOD	Two place vector containing the first and last date of the period, see SET_PERIOD command.
	WINDOW	Two place vector containing the beginning and end of the time window, see SET_WINDOW,
	ORDER	"LOG" or "EVENT".
	DIRECTION	"FORWARD" or "BACKWARD".
	TIMEOUT	Real value, the query time-out in seconds.
	ATTRIBUTES	Text vector, the attributes returned by the query.
	CONDITION	Text value, the query condition.
	STATUS	Integer value, SCIL status code.

## 10.14.33 HISTORY\_DATABASE\_MANAGER("QUERY", session, count [,start])

Performs a history database query.

'session'	Integer value returned by OPEN command.	
'count'	Integer value, the maximum number of events to be returned.	
'start'	Integer or vector value, the identifier of event to start the query at. Integer value 0 restarts the query. Default is 0.	
Value:	List:	
	STATUS	Integer value, SCIL status code.
	SUCCESS	Text value, indicating the success of the query:
	"DONE"	All the specified events were found.
	"MORE"	The number of events specified by 'count' were found. There may be more events to find.
	"TIMEOUT"	The query took a too long time and was interrupted. There may be more events to find.
	COUNT	Integer value, the number of the returned events.
	LAST	Vector value, the event identifier of the last processed event, or integer 0, if no events processed.
	ID	Vector value containing the identifiers of the returned events.
	DATA	List value containing the attributes of the returned events.

To continue a query, the event id returned by the previous query (attribute LAST) should be used as the argument 'start' of the subsequent query. The event identified by 'start' is not included in the result.

The attribute DATA of the function result contains the event data read from the database as a list. The attributes of the list are the ones specified by SET\_ATTRIBUTES command. History database information related to each event are described in the Application Objects manual. The value of each attribute is a vector of length COUNT. If an event does not have a certain attribute, the corresponding element in the vector has status PROF\_ATTRIBUTE\_DOES\_NOT\_EXIST. The list may be further processed with SELECT function by the event list dialog, if needed.

When a query is interrupted by TIMEOUT and the query order is EVENT, it is not guaranteed that the events returned by two subsequent queries are returned in exactly correct order. It is possible, that events contained in database files not yet processed should have been included in the results of the interrupted query.



The event identifiers returned by a query are valid only during the current period setting.

The following example reads the latest 20 events, then waits a second and reads the new ones. For clarity, error handling is omitted.

Example:

```
#LOCAL R, SESSION, FIRST, SECOND
R = HISTORY_DATABASE_MANAGER("OPEN")
SESSION = R.SESSION
R = HISTORY_DATABASE_MANAGER("SET_PERIOD", SESSION,
                             PACK_TIME(1998,4,1,0,0,0),CLOCK)
FIRST = HISTORY_DATABASE_MANAGER("QUERY",SESSION,20)
#IF FIRST.COUNT > 0 #THEN #BLOCK
    #PAUSE 1
    R = HISTORY_DATABASE_MANAGER("SET_ORDER",SESSION,"FORWARD")
    SECOND = HISTORY_DATABASE_MANAGER("QUERY",SESSION,100,FIRST.ID(1))
#BLOCK_END
```



The second query may miss a new event if its time-stamp is out of order.

#### 10.14.34 HISTORY\_DATABASE\_MANAGER("READ", session, event)

Reads all the attributes of an event.

History database information related to each event are described in the Application Objects manual.

'session'	Integer value returned by OPEN command.
'event'	Vector value, the event identifier of the event.
Value:	List:
	DATA        List value containing all the attributes of the event.
	STATUS      Integer value, SCIL status code.

#### 10.14.35 HISTORY\_DATABASE\_MANAGER("SET\_COMMENT", session, event, comment)

Sets the EX attribute of the specified event.

'session'	Integer value returned by OPEN command.	
'event'	Vector value, the event identifier of the event.	
'comment'	Text value, the comment.	
Value:	List:	
	STATUS	Integer value, SCIL status code.

## 10.14.36 HISTORY\_DATABASE\_MANAGER("WRITE", session, data)

Writes an event into the history database.

'data'	List value describing all the attributes of one event.	
'session'	Integer value returned by OPEN command.	
Value:	List.	
	STATUS	Integer value, SCIL status code.

This command writes an event into the history database. The attribute values to be written are given in the list argument 'data'.

Attribute HT is used to specify the history database file, into which the event is written. The file is created if it does not exist.

The following conventions are used to handle missing attributes:

1. Current time is used for missing HT, HM, HD and HQ.
2. HT, HM, HD and HQ values are used for missing ET, EM, ED and EQ.
3. All other missing numeric attributes are set to zero.
4. All missing text attributes are set empty.

## 10.14.37 Name hierarchy function

The NAME\_HIERARCHY function is used in database tools to create a tree view of process objects according to various naming attributes.

## 10.14.38 NAME\_HIERARCHY(names, order, syntax [, arg4 [, arg5]])

Constructs the tree hierarchy formed by given fully qualified hierarchical names.

'names'	A text vector, the fully qualified hierarchical names	
'order'	A text keyword specifying the order of branches and leaves in the result tree:	
	"SEPARATE"	Branch and leaf names are reported as separate text vectors.
	"INTERLEAVED"	Branch and leaf names are reported in one text vector, in alphabetical order.
'syntax'	A text value specifying the syntax of given names:	
	"OI"	The names are object identifiers (OI attribute values) of process objects.
	"ON"	The names are OPC item names (ON attribute values) of process objects.
	"ES"	The names are OPC event source names (ES attribute values) of process objects.

Table continues on next page

	any other text	The characters which are used as field delimiters of the given hierarchical names, for example "/".
'arg4'	Additional argument depending on the value of 'syntax':	
	When "OI":	An integer vector up to 5 elements which specifies the lengths of the fields of an object identifier.
	When "ON":	Not allowed
	When "ES":	The number of the application whose delimiter definitions are applied. This argument is optional: If omitted (or given as 0), current application is assumed. The application must be a local one. Note that the ES delimiters are defined in the OP attribute of the application.
'arg4' and 'arg5'	Optional text keyword arguments, which are used only when 'syntax' is not any of the predefined syntaxes:	
	"CASE_SENSITIVE"	The given names are case-sensitive. This is the default value and the value implicitly used by all predefined syntaxes.
	"CASE_INSENSITIVE"	The given names are case-insensitive.
	"ALLOW_DUPLICATES"	Duplicate names are allowed. This is the value implicitly used by syntaxes "OI" and "ES".
	"NO_DUPLICATES"	Duplicate names are flagged as errors. This is the default value and the value implicitly used by the "ON" syntax.
Value:	A list value containing two attributes:	
	ROOT	A node descriptor describing the root node of the tree. The structure of the node descriptor depends on the value of the 'order' argument, see below.
	ERRORS	An integer vector containing the indexes of the erroneous names in the argument vector 'names'.
	A node descriptor for the order "SEPARATE" is a list of following attributes:	
	BRANCH_COUNT	The number of branch nodes contained in the node.
	BRANCH_NAMES	A text vector, the names of the branches in alphabetical order. Omitted if BRANCH_COUNT is 0.
	BRANCHES	A list vector of length BRANCH_COUNT, the node descriptors of the branches. Omitted if BRANCH_COUNT is 0.
	LEAF_COUNT	The number of leaf names contained in the node. If duplicates are allowed, the number of leaves may be higher than this.
	LEAF_NAMES	A text vector, the leaf names in alphabetical order. Omitted if LEAF_COUNT is 0.
	LEAF_INDEXES	If duplicates are not allowed, the value is an integer vector listing the indexes of leaves in the 'names' vector. If duplicates are allowed, the value is a vector of vectors. Each element vector lists the indexes of leaves in the 'names' vector that share the name. Omitted if LEAF_COUNT is 0.
	A node descriptor for the order "INTERLEAVED" is a list of following attributes:	
	COUNT	The number of node names contained in the node.
	NAMES	A text vector, the node names in alphabetical order.

Table continues on next page

ROLES	A text vector containing the role of each node name: "BRANCH", "LEAF" or "BOTH".
BRANCHES	A list vector of length COUNT, the node descriptors of the branch nodes. If there are no branches, the attribute is omitted. Valid for roles "BRANCH" and "BOTH".
INDEXES	If duplicates are not allowed, the value is an integer vector listing the indexes of leaves in the 'names' vector. If duplicates are allowed, the value is a vector of vectors. Each element vector lists the indexes of leaves in the 'names' vector that share the name. Valid for roles "LEAF" and "BOTH". If there are no leaves, the attribute is omitted.

**Example 1:**

Syntax "ON", order "SEPARATE". The delimiter is dot, duplicates are not allowed.

```
#LOCAL NAMES = VECTOR("a.b.d", "a.b.c", "a.b", "e.f", "a.b.c")
#LOCAL TREE
TREE = NAME_HIERARCHY(NAMES, "SEPARATE", "ON")
; The value of the variable TREE is now:
; List 2
; ROOT List 4
; BRANCH_COUNT 2
; BRANCH_NAMES Vector 2
; (1) "a"
; (2) "e"
; BRANCHES Vector 2
; (1) List 6
; BRANCH_COUNT 1
; BRANCH_NAMES Vector 1
; (1) "b"
; BRANCHES Vector 1
; (1) List 4
; BRANCH_COUNT 0
; LEAF_COUNT 2
; LEAF_NAMES Vector 2
; (1) "c"
; (2) "d"
; LEAF_INDEXES Vector 2
; (1) 2
; (2) 1
; LEAF_COUNT 1
; LEAF_NAMES Vector 1
; (1) "b" ; "b" is both a leaf and a branch
; LEAF_INDEXES Vector 1
; (1) 3
; (2) List 4
; BRANCH_COUNT 0
; LEAF_COUNT 1
; LEAF_NAMES Vector 1
; (1) "f"
; LEAF_INDEXES Vector 1
; (1) 4
; LEAF_COUNT 0
; ERRORS Vector 1
; (1) 5 ;The 5th name "a.b.c" is a duplicate
```

**Example 2:**

Syntax "ON", order "INTERLEAVED". The delimiter is dot, duplicates are not allowed. The same data as in Example 1.

```
#LOCAL NAMES = VECTOR("a.b.d", "a.b.c", "a.b", "e.f", "a.b.c")
#LOCAL TREE
TREE = NAME_HIERARCHY(NAMES, "INTERLEAVED", "ON")
; The value of the variable TREE is now:
; List 2
; ROOT List 4
; COUNT 2
; NAMES Vector 2
;(1) "a"
;(2) "e"
; ROLES Vector 2
;(1) "BRANCH"
;(2) "BRANCH"
; BRANCHES Vector 2
;(1) List 5
; COUNT 1
; NAMES Vector 1
;(1) "b"
; ROLES Vector 1
;(1) "BOTH"
; BRANCHES Vector 1
;(1) List 4
; COUNT 2
; NAMES Vector 2
;(1) "c"
;(2) "d"
; ROLES Vector 2
;(1) "LEAF"
;(2) "LEAF"
; INDEXES Vector 2
;(1) 2
;(2) 1
; INDEXES Vector 1
;(1) 3
;(2) List 4
; COUNT 1
; NAMES Vector 1
;(1) "f"
; ROLES Vector 1
;(1) "LEAF"
; INDEXES Vector 1
;(1) 4
; ERRORS Vector 1
;(1) 5 ;The 5th name "a.b.c" is a duplicate
```

**Example 3:**

Syntax "OI", order "SEPARATE". Duplicates are allowed.

```
#LOCAL NAMES = VECTOR("Elgarose Transformer Q91",-  
                      "Elgarose Coles Valley Q1",-  
                      "Eastwick Incoming 110kV Q0",-  
                      "Elgarose Transformer Q91")  
#LOCAL SV15 = APL:BSV15  
#LOCAL OI = SV15.PROCESS_OBJECTS.OI  
#LOCAL TREE  
TREE = NAME_HIERARCHY(NAMES, "SEPARATE", "OI",-  
                      VECTOR(OI.LENGTH1, OI.LENGTH2,OI.LENGTH3))  
;The value of the variable TREE is now:  
;List 2  
; ROOT List 4  
; BRANCH_COUNT 2  
; BRANCH_NAMES Vector 2  
;(1) "Eastwick"  
;(2) "Elgarose"  
; BRANCHES Vector 2  
;(1) List 4  
; BRANCH_COUNT 1
```

```

; BRANCH_NAMES Vector 1
; (1) "Incoming 110kV"
; BRANCHES Vector 1
; (1) List 4
; BRANCH_COUNT 0
; LEAF_COUNT 1
; LEAF_NAMES Vector 1
; (1) "Q0"
; LEAF_INDEXES Vector 1
; (1) Vector 1
; (1) 3
; LEAF_COUNT 0
; (2) List 4
; BRANCH_COUNT 2
; BRANCH_NAMES Vector 2
; (1) "Coles Valley"
; (2) "Transformer"
; BRANCHES Vector 2
; (1) List 4
; BRANCH_COUNT 0
; LEAF_COUNT 1
; LEAF_NAMES Vector 1
; (1) "Q1"
; LEAF_INDEXES Vector 1
; (1) Vector 1
; (1) 2
; (2) List 4
; BRANCH_COUNT 0
; LEAF_COUNT 1
; LEAF_NAMES Vector 1
; (1) "Q91"
; LEAF_INDEXES Vector 2
; (1) Vector 1
; (1) 1
; (2) 4
; LEAF_COUNT 0
; LEAF_COUNT 0
; ERRORS Vector 0

```

### 10.14.39 Mapping functions

Mapping functions map the logical APL, PRI and STA numbers used by application SCIL programs to corresponding physical (base system) object numbers, and vice versa.

### 10.14.40 LOGICAL\_MAPPING(otype, number [, apl])

Maps physical (base system) object numbers to logical object numbers.

'otype'	A text keyword, the object type
	"APL" Application object
	"PRI" Printer object
	"STA" Station object
'number'	An integer or integer vector, the physical number(s)
'apl'	An integer, the application where the mapping is done. Optional, default = 0 (current application). External applications are supported.

Table continues on next page

Value:	A list with the following attributes:
STATUS	SCIL status code, 0 = OK
MAPPING	Integer vector containing the logical object numbers corresponding to given physical numbers.

The function uses the mapping attributes AP, PR and ST of the application object to do the mapping. In the resulting value, a zero is returned for each physical object that does not have a logical mapping in the application.

Example:

```
#SET APL:BST(1) = 101
#SET APL:BST(2) = 102
#SET APL:BST(3) = 103
STAS = (1, 101, 103)
UNITS = LOGICAL_MAPPING("STA", STAS))
; UNITS now equals to VECTOR(0, 1, 3)
```

#### 10.14.41 PHYSICAL\_MAPPING(otype, number [, apl])

Maps logical object numbers to physical (base system) object numbers.

'otype'	A text keyword, the object type
"APL"	Application object
"PRI"	Printer object
"STA"	Station object (unit)
'number'	An integer or integer vector, the logical number(s)
'apl'	An integer, the application where the mapping is done. Optional, default = 0 (current application). External applications are supported.
Value:	A list with the following attributes:
STATUS	SCIL status code, 0 = OK
MAPPING	Integer vector containing the physical object numbers corresponding to given logical numbers.

The function uses the mapping attributes AP, PR and ST of the application object to do the mapping. In the resulting value, a zero is returned for each logical object that does not map to any physical object.

Example:

```
#SET APL:BST(1) = 101
#SET APL:BST(2) = 102
#SET APL:BST(3) = 103
UNITS = (3, 1)
STAS = PHYSICAL_MAPPING("STA", UNITS)
; STAS now equals to VECTOR(103, 101)
```

### 10.15 Network Topology Functions

#### 10.15.1 NETWORK\_TOPOLOGY\_MANAGER(subfunction [, arg]\* [, apl])

Manages the network topology functionality of an application.

'subfunction'	Text keyword, the requested subfunction. The subfunctions are described in detail below.
'arg'	Up to 2 subfunction specific arguments.
'apl'	Integer, 0 ... 250. The logical application number. 0 = the own application. External applications are supported. Optional, default value is 0.
Value:	List: STATUS Integer value, SCIL status code. Subfunction specific return attributes.

In the following subfunction descriptions, the optional argument 'apl' is not repeated.

## 10.15.2 NETWORK\_TOPOLOGY\_MANAGER("SCHEMAS")

Lists the supported network topology schemas.

Value:	List: STATUS Integer value, SCIL status code. SCHEMAS Text vector, names of supported schemas. Currently, only the POWER schema is supported.
--------	---

## 10.15.3 NETWORK\_TOPOLOGY\_MANAGER("SCHEMA", schema)

Returns information about a schema.

'schema'	Text, the name of the schema.
Value:	List: STATUS Integer value, SCIL status code. NAME Text, the name of the schema. NT Text vector, the text identifiers used to localize the descriptions of numeric NT (Network Object State) attribute values. NS Text vector, the text identifiers used to localize the descriptions of numeric NS (Network Object Subtype) attribute values. LP Text vector, the text identifiers used to localize the descriptions of numeric LP (Loop State) attribute values.

## 10.15.4 NETWORK\_TOPOLOGY\_MANAGER("LEVELS", schema)

Lists the application specific (voltage) levels of the schema.

'schema'	Text, the name of the schema.
Value:	List: STATUS Integer value, SCIL status code. LV Real vector, the nominal (voltage) levels. MAX Real vector. Each MAX(i) defines the maximum level value (voltage) that is considered to belong to level i.

## 10.15.5 **NETWORK\_TOPOLOGY\_MANAGER("SET\_LEVELS", schema, levels)**

Sets the application specific (voltage) levels of the schema.

'schema'	Text, the name of the schema.
'levels'	List: LV Real vector, the nominal (voltage) level values. MAX Real vector. Each MAX(i) defines the maximum level value (voltage) that is considered to belong to level i.
Value:	List: STATUS Integer value, SCIL status code.

## 10.15.6 **NETWORK\_TOPOLOGY\_MANAGER("MODELS")**

Lists the imported network topology models of the application.

Value:	List: STATUS Integer value, SCIL status code. MODELS Text vector, names of imported models.
--------	---

## 10.15.7 **NETWORK\_TOPOLOGY\_MANAGER("MODEL", model)**

Returns information about a model.

'model'	Text, the name of the model.
Value:	List: STATUS Integer value, SCIL status code. NAME Text, the name of the model. SCHEMA Text, the schema of the model. GENERATOR Text, the name of the tool that generated the model. GENERATED_BY Name of the user who generated the model. GENERATION_DATE Time, the generation date of the model. BUILT_FROM Text vector, the names of process displays used to generate the model. VERSION Integer, the version number of the model format. COMMENT Text, any comment added by the generator of the model. IMPORTED_BY Name of the user who imported the model. IMPORT_DATE Time, the import date and time. RUNNING Boolean. See subfunctions START/STOP.

## 10.15.8 **NETWORK\_TOPOLOGY\_MANAGER("START", model), NETWORK\_TOPOLOGY\_MANAGER("STOP", model)**

Starts/stops calculation of a model.

'model'	Text, the name of the model.
Value:	List:
	STATUS Integer value, SCIL status code.

## 10.15.9 NETWORK\_TOPOLOGY\_MANAGER("VALIDATE", modeldata)

Validates a model.

'modeldata'	List, the complete description of the model in internal format.
Value:	List:
	STATUS Integer value, SCIL status code. If the model is not valid, the status is 2801 (TOPO_MODEL_CONTAINS_ERRORS)
	ERRORS Vector, descriptions of found errors
	WARNINGS Vector, descriptions of warnings

## 10.15.10 NETWORK\_TOPOLOGY\_MANAGER("IMPORT", modeldata)

Imports a new model or updates an existing model.

'modeldata'	List, the complete description of the model in internal format.
Value:	List:
	STATUS Integer value, SCIL status code. If the model is not valid, the status is 2801 (TOPO_MODEL_CONTAINS_ERRORS)

An existing model must be stopped before it can be re-imported.

## 10.15.11 NETWORK\_TOPOLOGY\_MANAGER("EXPORT", model)

Exports a model from the application

'model'	Text, the name of the model.
Value:	List:
	STATUS Integer value, SCIL status code.
	MODEL List, the complete description of the model in internal format.

## 10.15.12 NETWORK\_TOPOLOGY\_MANAGER("DELETE", model)

Deletes a model from the application.

'model'	Text, the name of the model.
Value:	List:
	STATUS Integer value, SCIL status code.

A model must be stopped before it can be deleted.

## 10.16 File handling functions

### 10.16.1 DATA\_MANAGER(function [,argument]\*)

SCIL database management.

'function'	A text keyword, the subfunction to be performed
'argument'	A subfunction specific list of other arguments
Value:	A list always containing the attribute:  STATUS      Integer value, SCIL status code.  Other subfunction specific attributes may be returned.

The structure and properties of SCIL databases are described in [Section 6.6](#).

Each subfunction is described in detail below.

### 10.16.2 DATA\_MANAGER("CREATE", file)

Creates a new SCIL database and opens it for use.

'file'	A text or byte string value, the name or tag of the file to be created. Extension SDB is recommended.
Value:	A list with attributes:  STATUS      An integer, the SCIL status code  HANDLE     An integer, a handle to the file to be used in subsequent calls of the function  Up to 10 databases may be concurrently open within a SCIL context.  VERSION    Integer 2 or 3, the version of the file format

### 10.16.3 DATA\_MANAGER("OPEN", file)

Opens an existing SCIL database.

'file'	A text or byte string value, the name or tag of the file to be opened
Value:	A list with attributes:  STATUS      An integer, the SCIL status code  HANDLE     An integer, a handle to the file to be used in subsequent calls of the function  Up to 10 databases may be concurrently open within a SCIL context.  VERSION    Integer 2 or 3, the version of the file format

### 10.16.4 DATA\_MANAGER("COPY", handle, new\_file [, version])

Makes a copy of an open SCIL database into another file.

'handle'	An integer, a handle to the source file
'new_file'	A text or byte string value, the name or tag of the new file
'version'	Integer value 2, 3 or 0

Table continues on next page

	Default value 0 (the newest version, currently equal to 3)	
Value:	A list with attributes:	
	STATUS	An integer, the SCIL status code
	FAILED	A text vector, the names of the sections that could not be copied

For compatibility, this function may be used to convert an old (Rev. 8.4.5) version 2 database to the new (Rev. 9.0) version 3 format for faster access, or vice versa.

## 10.16.5 DATA\_MANAGER("CLOSE", handle)

Closes a SCIL database when no longer used.

'handle'	An integer, a handle to the file
Value:	A list with one attribute:
	STATUS An integer, the SCIL status code

If a SCIL program does not close the file, it is automatically closed when the SCIL context of the program is destroyed.

## 10.16.6 DATA\_MANAGER("LIST\_SECTIONS", handle)

Lists the sections of a SCIL database in alphabetic order.

'handle'	An integer, a handle to the file
Value:	A list with attributes:
	STATUS An integer, the SCIL status code
	SECTIONS A text vector, the section names

## 10.16.7 DATA\_MANAGER("CREATE\_SECTION", handle, section)

Creates a new (empty) section in a SCIL database.

'handle'	An integer, the handle to the file
'section'	A text value, the name of the section
Value:	A list with one attribute:
	STATUS An integer, the SCIL status code

The value of the newly created section is an empty list.

## 10.16.8 DATA\_MANAGER("DELETE\_SECTION", handle, section)

Deletes a section, both the name and the contents, from SCIL database.

'handle'	An integer, the handle to the file
'section'	A text value, the name of the section
Value:	A list with one attribute:
	STATUS An integer, the SCIL status code

## 10.16.9 DATA\_MANAGER("GET", handle, section [,component]\*)

Reads data from SCIL database.

'handle'	An integer, the handle to the file
'section'	A text value, the name of the section
'component'	A text, an integer or a vector value, the component to be read
	Up to 5 'component' arguments may be given
Value:	A list with attributes:
	STATUS     An integer, the SCIL status code
	VALUE     Any type, the value read from the file

If no 'component' is specified, the entire section is read. The components, i.e. attributes of lists and indices or index ranges of vectors, are given in SCIL language syntax. Examples of valid component descriptions:

"ABC"	Attribute ABC
"(5)" or 5	The 5th element of a vector
"(1 .. 10)" or (1,10)	10 first elements of a vector
"(3 .. )" or (3,0)	The elements from the 3rd one to the last one
"ABC(2).DEF"	Attribute DEF of the second element of a (vector type) attribute ABC

The component description may be given as several arguments as well. The following examples are equivalent to the last example above.

"ABC", "(2)", "DEF"

"ABC", 2, "DEF"

Note, that only one component is read even if more than one 'component' arguments are given. These arguments are combined to locate the component deep in the SCIL data structure.

## 10.16.10 DATA\_MANAGER("PUT", handle, section, data [,component]\*)

Writes data to SCIL database.

'handle'	An integer, the handle to the file
'section'	A text value, the name of the section
'data'	Any value, the data to be written
'component'	A text, an integer or a vector value, the component to be written
	Up to 5 'component' arguments may be given.
Value:	A list with one attribute:
	STATUS     An integer, the SCIL status code

If no 'component' is specified, the entire section is written. See subfunction "GET" for description of 'component's.

If the specified component (attribute or vector element) does not exist in the database, it is created.

## 10.16.11 DATA\_MANAGER("DELETE", handle, section, [,component]\*)

Deletes a component from a SCIL database.

'handle'	An integer, the handle to the file
'section'	A text value, the name of the section
'component'	A text, an integer or a vector value, the component to be deleted Up to 5 'component's may be given.
Value:	A list with one attribute: STATUS An integer, the SCIL status code

If no 'component' is specified, the entire content (but not the name) of the section is deleted.  
See subfunction "GET" for description of components.

If the specified component is an attribute, it is deleted from its containing list.

If the specified component is an element of a vector, the element is deleted. The length of the vector will thus decrement by one.

If the specified component is a range of vector elements, the elements are deleted. The length of the vector will thus decrement by the number of deleted elements.

## 10.16.12 DELETE\_PARAMETER(file, section [,key])

Deletes a parameter from a parameter file.

'file'	Text or byte string (file tag). The name of the parameter file. See <a href="#">Section 6.5.1</a> for file naming.
'section'	Text. The name of the section.
'key'	Text. The key of the parameter to deleted. If 'key' is omitted from the argument list, the whole section is deleted.
Value:	A list with one attribute: STATUS An integer, the SCIL status code

The parameter files are described in [Section 6.5.2](#).

See also functions READ\_PARAMETER and WRITE\_PARAMETER.

## 10.16.13 FILE\_LOCK\_MANAGER(function, file)

Locks and unlocks files.

This function is used to temporarily lock a file for use of only one SCIL context. This function is mainly needed by tools (such as Dialog Editor) to prevent simultaneous modification of same data by two users. All the SCIL tools accessing the file should use FILE\_LOCK\_MANAGER to synchronize the access to the file, locking a file with FILE\_LOCK\_MANAGER does not prevent accessing of the file by READ\_TEXT, WRITE\_TEXT or other SCIL functions.

'function'	Text value, the function to be performed:
"LOCK"	Lock the file for exclusive use.
"UNLOCK"	Unlock the file.

Table continues on next page

	"BREAK"	Break the lock.
'file'		Text value containing the name of the file to be locked in OS dependent format (see PARSE_FILE_NAME function to obtain OS file names).
Value:	A text value containing:	
	"OK"	Function successfully performed.
	"LOCKED"	LOCK failed because of an existing lock.
	"BROKEN"	UNLOCK detected that the file was not locked, i.e. broken by somebody else.
	"INVALID"	The 'file' argument is invalid. This means that the name is not a valid file name or the directory in the name does not exist or the access to it is denied.

**Notes:**

- The file does not have to exist, actually 'file' argument is used as a unique identifier of the lock. The argument is used a seed for a lock file name, which is generated by prefixing the file name in 'file' by "\_L\_". For example, if 'file' is "C:\DATA\ABC.DAT", then the lock file name used is "C:\DATA\\_L\_ABC.DAT". Therefore, the file name should not be extremely long.
- Two different 'file' arguments may refer to one and same lock. For example, if a logical drive K points to directory "C:\DATA", then 'file' arguments "C:\DATA\ABC.DAT" and "K:\ABC.DAT" denote the same lock.
- When a SCIL context is destroyed, all the locks it holds are automatically released by the base system software.
- The function "BREAK" should be used with care. It should be used only after a program crash or other such failure which has left the file locked.

## 10.16.14 KEYED\_FILE\_MANAGER(function, file [,output\_file [,key\_size] [,version]])

File maintenance function.

'function'	Text: "INFORMATION", "COMPACT", or "REBUILD". The operation to be performed on the file (see below):										
'file'	Text or byte string (file tag). The input file name. See <a href="#">Section 6.5.1</a> for file naming.										
'output_file'	Text or byte string (file tag). The output file name (functions COMPACT and REBUILD only). See <a href="#">Section 6.5.1</a> for file naming.										
'key_size'	Integer value, the size of the record key (function REBUILD only).										
'version'	Integer value 1 or 2, the version number of the output file format (functions COMPACT and REBUILD only). Default value is the version of the input file. The maximum size of version 1 files is 32 MB. Version 2 files, introduced in MicroSCADA rev. 8.4.4, do not have such a restriction.										
Value:	<p>List value containing various information depending on the function argument.</p> <p>The 'function' value "INFORMATION" returns the following attributes in the result list:</p> <table> <tr> <td>STATUS</td> <td>The status of the operation.</td> </tr> <tr> <td>VERSION</td> <td>The file format version number, currently 1 or 2.</td> </tr> <tr> <td>USED_BLOCKS</td> <td>Number of 512-byte blocks (version 1 files) or 4096-byte blocks (version 2 files) allocated for the file.</td> </tr> <tr> <td>KEY_SIZE</td> <td>The length of the record key.</td> </tr> <tr> <td>INDEX_LEVELS</td> <td>The number of index levels used by the index tree of the file.</td> </tr> </table> <p>The attributes VERSION, USED_BLOCKS and INDEX_LEVELS are present only if STATUS = 0.</p>	STATUS	The status of the operation.	VERSION	The file format version number, currently 1 or 2.	USED_BLOCKS	Number of 512-byte blocks (version 1 files) or 4096-byte blocks (version 2 files) allocated for the file.	KEY_SIZE	The length of the record key.	INDEX_LEVELS	The number of index levels used by the index tree of the file.
STATUS	The status of the operation.										
VERSION	The file format version number, currently 1 or 2.										
USED_BLOCKS	Number of 512-byte blocks (version 1 files) or 4096-byte blocks (version 2 files) allocated for the file.										
KEY_SIZE	The length of the record key.										
INDEX_LEVELS	The number of index levels used by the index tree of the file.										

Table continues on next page

The 'function' value "COMPACT" returns the same attributes as the 'function' value "INFORMATION" (the values are those before the compacting) and in addition the following attribute:

RECORDS                    Number of records copied.

The 'function' value "REBUILD" returns the same attributes as the 'function' value "COMPACT" and in addition the following attribute:

DUPLICATES                A text vector containing the key values that were found duplicated in the source file. The corresponding records in the output file may contain obsolete or otherwise bad data, they should be checked. Max. 1000 duplicate keys are reported.

If the status block of the file was corrupted, attributes VERSION, USED\_BLOCKS and INDEX\_LEVELS are not returned.

This function converts SYS600 keyed files from a format to another, saves corrupted files and does some other file maintenance. It can handle one file at a time, the 'file' argument, and provides the following operations selected by the 'function' argument:

"INFORMATION"	Returns some information of the file.
"COMPACT"	Compacts the file by rewriting the records in their key order. After the compacting, the file is smaller and faster to access. The off-line program REORG was previously used for this purpose.
"REBUILD"	Reconstructs the file by scanning and rewriting the data blocks of the file, ignoring the index blocks. This function should be used if the internal structure of the file is corrupted, for example if status 5015 (FILE_INCONSISTENT) is returned when the file is accessed. The file may not be used by SYS600 when this function is performed. The optional argument 'key_size' overrides the key size read from the status block of the file. It should be used only if the status block of the file is corrupted. The corruption is indicated by status code 5016 (FILE_INVALID_KEY_SIZE) when attempting this function without the key_size argument, or the same status code when trying to access the output file generated by this function without the key_size argument. The correct value of this argument is obtained by applying the "INFORMATION" function to an uncorrupted file of the same type.

See [Section 6.5.4](#) for more information about keyed files.

Example:

```
@RESULT = KEYED_FILE_MANAGER("COMPACT", -
    "/APL/TIPPERARY/APL_/APL_PROCES.PRD", -
    "/APL/TIPPERARY/APL_V2/APL_PROCES.PRD", -
    2)
; Converts the process database of application TIPPERARY to file format 2.
```

## 10.16.15 PARSE\_FILE\_NAME(name [,file])

Converts SCIL path names and file names to operating system file names.

'name'	Text value: directory or path name, either 1) SCIL path name, for example, "PICT" 2) SCIL directory name, for example, "/APL/TEST/PICT" or "PICT/" 3) OS dependent directory name, for example, "C:\SC\APL\TEST\PICT"
'file'	Text value: optional file name, for example, "station.pic"
Value:	Text value containing the path name of the file (or directory) in OS dependent format, for example, "C:\SC\APL\TEST\PICT\STATION.PIC" (or "C:\SC\APL\TEST\PICT"). Returns "", if there is an error in arguments.

Notes:

- The directory name in SCIL format may be given with or without the trailing "/" (see however note 3).
- The OS dependent directory name may be given with or without the trailing OS dependent delimiter (for example, "\").
- In SCIL, depending on the context, "PICT" may mean either the path PICT or the directory "/APL/xxx/PICT". To resolve the ambiguity, "PICT" is used for the path and "PICT/" for the directory.
- If a path name is given as 'name' and file is not given, the function returns the first directory in the path.
- If a path name is given as 'name' and 'file' exists, the complete file path name of the found file is returned.
- If a path name is given as 'name' and 'file' does not exist, the returned path name contains the first directory of the logical path, i.e. the returned value is the complete would-be name of the file if created.
- The arguments of the function are case-insensitive. The case of the returned value is OS dependent. In NT, upper case string is returned.

## 10.16.16 PATH(name)

The directories contained in a logical path.

'name'	Text expression, the name of a logical path.
Value:	Text vector containing the names (in an operating system dependent format) of directories that make up the logical path.

See #PATH command in [Section 9.2.4](#) for details of logical paths.

## 10.16.17 PATHS(level)

The logical paths defined on a specified level.

'level'	Integer value 0 to 3 specifying the requested level in path hierarchy:
0	System paths
1	Application paths
2	Process specific paths (for example, monitor paths)
3	Temporary paths
Value:	Text vector containing the names of the defined logical paths on the requested level.

See #PATH command in [Section 9.2.4](#) for details of logical paths.

## 10.16.18 READ\_BYTES(file [,start [,length]])

Reads a binary file.

'file'	Text or byte string (file tag). The name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'start'	Positive integer value, defaults to 1. Specifies the byte position within the file where to start reading.
'length'	Non-negative integer value. Specifies the maximum number of bytes to be read. Default value = 8 388 600 (max. byte string length).
Return value:	Byte string containing the read data.

Using this function any file may be read as a sequential binary file, i.e. an unstructured string of bytes. It is usually used to import data generated by some external application or to read a compiled SCIL program from a file

See also function [WRITE\\_BYTES](#).

## 10.16.19 **READ\_COLUMNS(file, pos, width [,start [,count]])**

Reads a text file as columns.

The function reads a text file as columns and stores them in a vector where each element is a text vector containing the text of one column.

'file'	Text or byte string (file tag). The name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'pos'	Integer vector. Specifies the start positions of the columns to be read (1 ... 65 535). The positions do not have to be in ascending order.
'width'	Integer vector. The 'width' argument must be of the same length as the 'pos' argument. Specifies the column widths (1 ... 65 535). The columns are allowed to overlap.
'start'	Positive integer value, defaults to 1. Specifies the line number within the file where to start reading.
'count'	Non-negative integer value, defaults to 10 000. Specifies the maximum number of lines to be read.
Value:	A vector of the same length as 'pos' and 'width'. Each element of the vector is a text vector containing the text of one column.

The function supports Unicode files, UTF-8 and UTF-16 Byte Order Marks (BOM's) are recognized.

See also function [WRITE\\_COLUMNS](#).

## 10.16.20 **READ\_PARAMETER(file, section, key [,default])**

Reads a parameter from a parameter file.

'file'	Text or byte string (file tag). The name of the parameter file. See <a href="#">Section 6.5.1</a> for file naming.
'section'	Text. The name of the section.
'key'	Text. The key of the parameter.
'default'	Text. The value to be returned if 'section' or 'key' does not exist in the specified file.
Value:	A list with the following two attributes:
	STATUS            The status code of the read operation.
	VALUE            The value of the parameter. Returned only if STATUS = 0.

The parameter files are described in [Section 6.5.2](#).

If the given section or key does not exist, error code SCIL\_SECTION\_DOES\_NOT\_EXIST or SCIL\_KEY\_DOES\_NOT\_EXIST is returned in the STATUS attribute.

See also functions WRITE\_PARAMETER and DELETE\_PARAMETER.

Example:

```
#LOCAL PORT = READ_PARAMETER("C:\WINNT\WIN.INI", "MCILAU", "UDP PORT")
```

## 10.16.21 READ\_TEXT(file [,start [,number]])

Reads a text file.

'file'	Text or byte string (file tag). The file name. See <a href="#">Section 6.5.1</a> for file naming.
'start'	Positive or negative integer. If positive, it is the number of the first line to be read from the text file counted from the beginning of the file (1 = the first line). If negative, it is the number of the last line to be read counted from the end of the file (-1 = the last line). Default: 1.
'number'	Integer, 0 ... 1 000 000. The number of lines to be read from the file. 0 = nothing is read. Default: 1000.
Value:	A text vector containing the lines read from the file.

If 'start' is omitted, reading starts from the beginning of the file. If 'number' is omitted, reading is performed to the end of file, or until 1000 lines have been read.

The function supports Unicode files, UTF-8 and UTF-16 Byte Order Marks (BOM's) are recognized.

See also function WRITE\_TEXT.



For compatibility reasons, this function does not support lines longer than 255 characters. Longer lines are silently truncated to 255 characters. This function is now more or less obsolete, use the function TEXT\_READ instead.

Example:

```
! SHOW DIRECTIVE READ_TEXT("DIRECTIVE.TXT")
;The contents of the file DIRECTIVE.TXT are shown in the window DIRECTIVE.

#LOCAL ABC = LIST(IU = 1,-
    IN = READ_TEXT("ABC.TXT"))
#CREATE ABC:C = ABC
;Command procedure ABC is created. Its SCIL program is read from a file.

READ_TEXT("FILE", -1, 100) ;reads the last 100 lines
READ_TEXT("FILE", -101, 100) ;reads the preceding 100 lines.
```

## 10.16.22 REP\_LIB(name)

The files contained in a logical representation library.

'name'	Text expression, the name of the logical representation library.
Value:	Text vector containing the names (in an operating system dependent format) of library files that make up the logical library.

See #REP\_LIB command in [Section 9.2.4](#)

## 10.16.23 REP\_LIBS(level)

The logical library names defined on a specified level.

'level'	Integer value 0 to 3 specifying the requested level in library hierarchy:
0	System libraries
1	Application libraries
2	Process specific libraries (for example, monitor libraries)
3	Temporary libraries
Value:	Text vector containing the names of the defined logical representation libraries on the requested level.

See #REP\_LIB command in [Section 9.2.4](#) for details of logical representation libraries.

## 10.16.24 SHADOW\_FILE(file\_name [, "DELETED"])

Queues a file for shadowing.

'file_name'	Text or byte string (file tag). The name of the file. See <a href="#">Section 6.5.1</a> for file naming.
"DELETED"	Optional keyword "DELETED".
Value:	A list value containing the following attributes:
STATUS	status code (0 = OK).
QUEUED	Boolean value indicating whether the file was queued for shadowing.

The function first locates the file. The status is returned in attribute STATUS. Then, it checks whether the file belongs to an application that is currently shadowed. If yes, the file is queued for shadowing (the function does not wait for the completion of shadowing) and TRUE is returned in attribute QUEUED. If not, FALSE is returned and nothing is done.

If optional keyword "DELETED" is specified, the file is expected to be already deleted from the local system. In this case the file delete operation is only queued for shadowing.

This function should be used if a file is created in an application directory (a directory below the directory "/APL/application") using a non-SYS600 program, for example, by a tool that copies files using operating system utilities.

## 10.16.25 TEXT\_READ(file [,start [,number]])

Reads a text file or a part of it.

'file'	Text or byte string (file tag). The file name. See <a href="#">Section 6.5.1</a> for file naming.
'start'	Positive or negative integer. If positive, it is the number of the first line to be read from the text file counted from the beginning of the file (1 = the first line). If negative, it is the number of the last line to be read counted from the end of the file (-1 = the last line). Default: 1.
'number'	Integer, 0 ... 2 000 000. The number of lines to be read from the file. 0 = nothing is read. Default: 10 000.
Value:	A list with the following attributes:

Table continues on next page

STATUS	SCIL status code, 0 = OK
TEXT	Text vector containing the read data (up to 2 000 000 lines, each line up to 65 535 characters)
LONGEST	Integer, the index of the longest line in TEXT
MORE	Boolean, TRUE if there are more lines to be read, otherwise FALSE

If 'start' is omitted, reading starts from the beginning of the file. If 'number' is omitted, reading is performed to the end of file, or until 10 000 lines have been read.

The attributes TEXT, LONGEST and MORE are returned only when STATUS = 0.

The function supports Unicode files, UTF-8 and UTF-16 Byte Order Marks (BOM's) are recognized.

## 10.16.26 WRITE\_BYTES(file, data [,append])

Writes a binary file.

'file'	Text or byte string (file tag). The name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'data'	Byte string value containing the data to be written.
'append'	Integer value 0 or 1, defaults to 0. If 0, a new file is created. If 1, the data are appended to the file, if it already exists.
Value:	Integer value. The status code of file write operation, 0 if OK.

Using this function a byte string value may be written as a sequential binary file, i.e. an unstructured string of bytes. It is usually used to export data to an external application or to store a compiled SCIL program in a file.

See also function READ\_BYTES.

## 10.16.27 WRITE\_COLUMNS(file, pos, width, data [,append] [,encoding])

Writes a text file as columns.

'file'	Text or byte string (file tag). The name of the file. See <a href="#">Section 6.5.1</a> for file naming.
'pos'	Integer vector. Specifies the start positions of the columns to be written (1 ... 65 535). The positions do not have to be in ascending order.
'width'	Integer vector. Must be of the same length as pos. Specifies the column widths (1 ... 65 535). The columns are allowed to overlap. If they do, a column earlier in the list will be covered by the later one. Any data written past character position 65 535 are lost. Possible gaps between the columns are filled with space characters.
'data'	Vector value of the same length as 'pos' and 'width'. Each element of the vector is a text vector containing the text of one column. The element vectors must be of the same length.
'append'	Integer value 0 or 1, defaults to 0. If 0, a new file is created. If 1, the data are appended to the file, if it already exists.
'encoding'	Text keyword, either "UTF" or "ANSI":
"UTF"	The file is written in UTF-8 encoding with a BOM (Byte Order Mark). This is the default value.

Table continues on next page

"ANSI" The file is written in extended ANSI encoding using the ACP (Active Code Page) of the operating system. This value is used when the file is later read by an application that is not Unicode aware.

When an existing file is appended, this argument is ignored. The encoding of the file is reserved.

Value: Integer value. The status of the write operation, 0 if OK.

Arguments 'append' and 'encoding' may be given in any order.

See also function READ\_COLUMNS.

## 10.16.28 WRITE\_PARAMETER(file, section, key, value [, encoding])

Writes a parameter into a parameter file.

'file' Text or byte string (file tag). The name of the parameter file. See [Section 6.5.1](#) for file naming.

'section' Text. The name of the section.

'key' Text. The key of the parameter.

'value' Text. The value to be assigned to the parameter.

'encoding' Text keyword, either "UTF" or "ANSI":  
"UTF" The file is written in UTF-8 encoding with a BOM (Byte Order Mark).

"ANSI" The file is written in extended ANSI encoding using the ACP (Active Code Page) of the operating system. This value is used when the file is later read by an application that is not Unicode aware. This is the default value.

Return value: A list with one attribute:

STATUS The status code of the write operation.

The file and/or the section is created if it does not exist. The parameter files are described in [Section 6.5.2](#).

Parameter file lines may be up to 65 535 characters long. If a longer line is written, the following error raises: SCIL\_PARAMETER\_FILE\_LINE\_TOO\_LONG.

Because the spaces before and after the equal sign are insignificant, it is not possible to write a key value with a leading space character.

See also functions READ\_PARAMETER and DELETE\_PARAMETER.

## 10.16.29 WRITE\_TEXT(file, text [,append] [,encoding])

Writes a text file.

'file' Text or byte string (file tag). The name of the file. See [Section 6.5.1](#) for file naming.

'text' A text vector containing the text to be written.

'append' Integer value 0 or 1, defaults to 0. If 0, a new file is created. If 1, the data are appended to the file, if it already exists.

'encoding' Text keyword, either "UTF" or "ANSI":

Table continues on next page

"UTF"	The file is written in UTF-8 encoding with a BOM (Byte Order Mark). This is the default value.
"ANSI"	The file is written in extended ANSI encoding using the ACP (Active Code Page) of the operating system. This value is used when the file is later read by an application that is not Unicode aware.
Value:	An integer. The status code of the file write, 0 = OK.

Arguments 'append' and 'encoding' may be given in any order.

See also functions TEXT\_READ and READ\_TEXT.

Example:

```
#LOCAL S = WRITE_TEXT("A", V, 1)
#IF S == 0 #THEN !SHOW INFO "File successfully appended"
#ELSE !SHOW INFO "Append failed by " + STATUS_CODE_NAME(S)
```

## 10.17 File management functions

File management functions implement the handling of drives (or disk-like devices), directories and files in an operating-system independent way.

This function family consists of functions DRIVE\_MANAGER, DIRECTORY\_MANAGER, FILE\_MANAGER and a number of auxiliary functions named FM\_\*.

Used terminology:

Drive name	Printable drive name in OS dependent format. In Windows, one-letter name A, B, ... or a UNC name. Lower case letters accepted as a function argument.
Directory name	Printable directory name. In Windows, any valid file name. Returned in case stored by the file system, case-insensitive as a function argument.
File name	Printable file name. In Windows, any valid file name. Returned in case stored by the file system, case-insensitive as a function argument. By convention, file name is divided into the proper file name and the extension separated by a period. If there are several periods in the name, the last one is considered as the name/extension separator.
Absolute path	Printable absolute directory or file path in OS dependent format. Absolute path contains the drive name (or uses the default drive) and all the intermediate directory names to uniquely identify a directory or a file. In Windows, the syntax of an absolute path is [drive:]\\[directory_name]\\*name.
Relative path	Printable relative directory or file path in OS dependent format. Given a root directory, relative path contains intermediate directory names (if any) to uniquely identify a directory or a file within the root directory. The absolute path of a file is obtained by combining the absolute path of the root directory and the relative path of the file. In Windows, the syntax of a relative path is [directory_name]\\*name.
SCIL name	File (or directory) name given in operating system independent format used by other SCIL commands and functions, see PARSE_FILE_NAME function for details. Examples: "PATH/FILE.EXT", "/SYS/ACTIVE/SYS_/SYS_BASCON.COM", "/LIB"
Drive tag	An OS independent identifier of a drive represented as a byte string data in SCIL.
Absolute directory tag	An OS independent representation of the absolute path of a directory. A drive tag is a valid absolute directory tag referring to the root directory of the drive.

Table continues on next page

Relative directory tag	An OS independent representation of the relative path of a directory.
Absolute file tag	An OS independent representation of the absolute path of a file.
Relative file tag	An OS independent representation of the relative path of a file. Filters used when browsing directories may contain following wildcard characters:
*	Matches with any character string including null string.
?	Matches with any single character, at the end of name or extension it matches also null character.
%	Matches with any single character.  An example: sysm?.exe? matches with sysm.exe, but sysm%.exe% does not.
	Operation of the filters is comparable to the Windows Find Files or Folders operation.

## 10.17.1 Calling syntax

The common calling syntax of DRIVE\_MANAGER, DIRECTORY\_MANAGER and FILE\_MANAGER is  
**FUNCTION\_NAME(command [,argument]\*)**

The first argument 'command' is a text keyword that selects the requested subfunction. The keyword is case-insensitive. In the descriptions of subfunctions below, the 'command' argument is represented as an upper-case text constant. Any valid SCIL expression resulting to a valid keyword value will do, of course.

The auxiliary functions do not follow this convention.

Example listing all files in the given directory:

```
TAGS = FM_DIRECTORY("c:\temp")
NAMES = FM_REPRESENT(FILE_MANAGER("LIST", TAGS))
;NAMES is a vector containing the filenames of the directory "c:\temp".
```

## 10.17.2 Compatibility

The functions in this section all use abstract tags instead of a textual names as file identifiers.

File handling commands ([Section 9.2.5](#)) and file handling functions ([Section 10.16](#)) accept a file tag argument as the identifier of the file to enable their use in conjunction with file management functions.

## 10.17.3 DRIVE\_MANAGER

A drive is the root of a file hierarchy. Depending on the operating system, a drive may correspond to a physical disk-like device or it may consist of several physical devices or it may be a partitioning of a physical device or it may map to a directory of another drive. In Windows, a drive corresponds to a Windows logical drive (labelled A, B, and so on) or an UNC (Universal Naming Convention) name of the form \\servername\sharename.

The following commands are recognized by DRIVE\_MANAGER:

- LIST
- EXISTS
- GET\_DEFAULT
- GET\_ATTRIBUTES

## 10.17.4 DRIVE\_MANAGER("LIST")

Returns the drives available in the system.

Value: Vector value containing the drive tags of the available drives.

## 10.17.5 DRIVE\_MANAGER("EXISTS", drive)

Checks the existence of one or more drives.

'drive' Drive tag or a vector of drive tags to be checked.

Value: Boolean or boolean vector value indicating whether the drive(s) exist or not.

## 10.17.6 DRIVE\_MANAGER("GET\_DEFAULT")

Returns the default drive, i.e. the drive assumed if an absolute path does not contain the drive.

Value: Byte string value containing the tag of the default drive.

## 10.17.7 DRIVE\_MANAGER("GET\_ATTRIBUTES", tag)

Returns some information from drives.

'tag' The drive tag or a vector of drive tags of interest.

Value: A list of following attributes:

STATUS Integer or integer vector value, the status code(s) of the query.

FAILURES Integer value containing the number of failed queries. If FAILURES == 0, STATUS contains all zeroes.

TYPE Text or text vector value:  
"FIXED"  
"REMOVABLE"  
"CDROM"  
"NETWORK"  
"RAM"  
"SHARED"  
"UNKNOWN"

CAPACITY Integer or integer vector value: The total capacity of the drive in kilobytes. 0 is returned for unavailable device of TYPE "REMOVABLE".

FREE Integer or integer value: The unused capacity of the drive in kilobytes. 0 is returned for unavailable device of TYPE "REMOVABLE".

## 10.17.8 DIRECTORY\_MANAGER

The following commands are recognized by DIRECTORY\_MANAGER:

- LIST
- CREATE
- DELETE
- DELETE\_CONTENTS
- EXISTS
- COPY
- COPY\_CONTENTS

- MOVE
- RENAME
- GET\_ATTRIBUTES

### **10.17.9 DIRECTORY\_MANAGER("LIST", directory [,filter [,recursion] [,hidden]]])**

Lists the directories contained in a given directory.

'directory'	Absolute directory tag of the directory whose contents are listed.
'filter'	Text value: The filter for the directory names to be listed. May contain wildcard characters * , % and ?. Default value is "" (no filter).
'recursion'	Text keyword "RECURSIVE" or "NON_RECURSIVE". Recursive listing means that whole directory hierarchy rooted in 'root' is listed. Default value is "NON_RECURSIVE".
'hidden'	Text keyword "EXCLUDE_HIDDEN" or "INCLUDE_HIDDEN". Specifies whether hidden directories are listed or not. Default value is "EXCLUDE_HIDDEN"
Value:	Vector value containing the relative directory tags of the directories found.

Arguments 'recursion' and 'hidden' may be given in any order.

### **10.17.10 DIRECTORY\_MANAGER("CREATE", directory [,recursion])**

Creates a directory or a hierarchy of directories.

'directory'	Absolute directory tag of the directory to be created.
'recursion'	Text keyword "RECURSIVE" or "NON_RECURSIVE". Recursive creation means that all missing directories contained in 'directory' are created. Default value is "NON_RECURSIVE".
Value:	Integer value: The status code of the operation.

### **10.17.11 DIRECTORY\_MANAGER("DELETE", directory)**

Deletes one or more directories and all the directories and files contained in them.

'directory'	Absolute directory tag or a vector of tags to be deleted.
Value:	List value with following attributes:
OK	Boolean value, TRUE if successful.
FAILED	A vector value containing relative directory or file tags of directories and files not deleted. Missing if OK = TRUE.
STATUS	An integer vector containing the status codes of failed deletions. Missing if OK = TRUE.

### **10.17.12 DIRECTORY\_MANAGER("DELETE\_CONTENTS", directory [,filter [,subdirectories]])**

Deletes files and directories contained in a given directory.

'directory'	Absolute directory tag of the directory whose contents are deleted.
'filter'	Text value: The filter for the file and directory names to be deleted. May contain wildcard characters * , % and ?. Default value is "" (no filter).
'subdirectories'	Text keyword "INCLUDE_DIRECTORIES" or "OMIT_DIRECTORIES". Specifies whether the subdirectories are deleted or not. If subdirectories are deleted, the filter is applied to the name of subdirectories (not to the contained files). Default value is "OMIT_DIRECTORIES".
Value:	List value with following attributes:
OK	Boolean value, TRUE if successful.
FAILED	A vector value containing relative directory or file tags of directories and files not deleted. Missing if OK = TRUE.
STATUS	An integer vector containing the status codes of failed deletions. Missing if OK = TRUE.

### 10.17.13 DIRECTORY\_MANAGER("EXISTS", directory)

Checks the existence of one or more directories.

'directory'	Absolute directory tag or a vector of tags to be checked.
Value:	Boolean or boolean vector value indicating whether the directories exist or not.

### 10.17.14 DIRECTORY\_MANAGER("COPY", source, target)

Copies a directory and all its contents into a new directory.

'source'	Absolute directory tag of the source directory.
'target'	Absolute directory tag of the target directory, which is created by COPY.
Value:	List value with following attributes:
OK	Boolean value, TRUE if successful.
FAILED	A vector value containing relative directory or file tags of directories and files not copied. Missing if OK = TRUE.
STATUS	An integer vector containing the status codes of failed copies. Missing if OK = TRUE.

### 10.17.15 DIRECTORY\_MANAGER("COPY\_CONTENTS", source, target [,filter [,subdirectories [,overwrite]]])

Copies the files of a directory into another directory. Optionally, the subdirectories are recursively copied as well.

'source'	Absolute directory tag of the source directory.
'target'	Absolute directory tag of the target directory (not created by COPY_CONTENTS).
'filter'	Text value: The filter for the file and directory names to be copied. May contain wildcard characters * , % and ?. Default value is "" (no filter).
'subdirectories'	Text keyword "INCLUDE_DIRECTORIES" or "OMIT_DIRECTORIES". Specifies whether the subdirectories are copied or not. If subdirectories are copied, the filter is applied to the name of subdirectories (not to the contained files). Default value is "OMIT_DIRECTORIES".

Table continues on next page

'overwrite'	Text keyword "OVERWRITE" or "DONT_OVERWRITE". Specifies whether an existing file in target directory is overwritten or not. Default value is "DONT_OVERWRITE"
Value:	List value with following attributes:
OK	Boolean value, TRUE if successful.
FAILED	A vector value containing relative directory or file tags of directories and files not copied. Missing if OK = TRUE.
STATUS	An integer vector containing the status codes of failed copies. Missing if OK = TRUE.

Keyword arguments 'subdirectories' and 'overwrite' may be given in any order.

## 10.17.16 DIRECTORY\_MANAGER("MOVE", directory, target)

Moves a directory to another directory.

'directory'	Absolute directory tag of the directory to be moved.
'target'	Absolute directory tag of the directory to become the new parent directory.
Value:	List value with following attributes:
STATUS	Integer value, the status code of the operation.
NEW_TAG	New tag for the moved directory (if STATUS = 0)

## 10.17.17 DIRECTORY\_MANAGER("RENAME", directory, name)

Renames a directory.

'directory'	Absolute directory tag of the directory to be renamed.
'name'	Text value, the new directory name.
Value:	List value with following attributes:
STATUS	Integer value, the status code of the operation.
NEW_TAG	New tag for the moved directory (if STATUS = 0)

## 10.17.18 DIRECTORY\_MANAGER("GET\_ATTRIBUTES", directory)

Returns attribute information from one or more directories.

'directory'	Absolute directory tag or vector of directory tags of interest.
value	List value containing the following attributes:
STATUS	Integer or integer vector value containing the status of each query.
READ_ONLY	Boolean or boolean vector value.
SYSTEM	Boolean or boolean vector value, TRUE if exclusively used by the OS.
HIDDEN	Boolean or boolean vector value, TRUE if a 'hidden' directory.
ARCHIVE	Boolean or boolean vector value, TRUE if 'archive' attribute is set in the directory.
COMPRESSED	Boolean or boolean vector value, TRUE if a compressed directory.
ENCRYPTED	Boolean or boolean vector value, TRUE if an encrypted directory.
FAILURES	Integer value telling the number of failed queries.

## 10.17.19 FILE\_MANAGER

The following commands are recognized by FILE\_MANAGER:

- LIST
- DELETE
- EXISTS
- COPY
- MOVE
- RENAME
- GET\_ATTRIBUTES

## 10.17.20 FILE\_MANAGER("LIST", directory [,filter [,recursion] [, hidden]])

Lists the files contained in a given directory.

'directory'	Absolute directory tag of the directory whose contents are listed.
'filter'	Text value: The filter for the file names to be listed. May contain wildcard characters * % and ?. Default value is "" (no filter).
'recursion'	Text keyword "RECURSIVE" or "NON_RECURSIVE". Recursive listing means that all files in the directory hierarchy rooted in 'directory' is listed. Default value is "NON_RECURSIVE".
'hidden'	Text keyword "EXCLUDE_HIDDEN" OR "INCLUDE_HIDDEN". Specifies whether hidden files are listed or not. Default value is "EXCLUDE_HIDDEN"
Value:	Vector value containing the relative file tags of the files found.

Arguments 'recursion' and 'hidden' may be given in any order.

## 10.17.21 FILE\_MANAGER("DELETE", file)

Deletes one or more files.

'file'	Absolute file tag or a vector of file tags to be deleted.
Value:	A list value containing the following attributes:
DELETED	Integer value containing the number of deleted files.
FAILED	Integer value containing the number of failed deletions.
STATUS	Integer or vector value containing the status code for each deletion.

## 10.17.22 FILE\_MANAGER("EXISTS", file)

Checks the existence of one or more files.

'file'	Absolute file tag or a vector of file tags to be checked.
Value:	Boolean or a boolean vector value indicating whether the file exists or not.

## 10.17.23 FILE\_MANAGER("COPY", source, target [,overwrite])

Copies the contents of a file to another file.

'source'	Absolute file tag of the source file.
'target'	Absolute file tag of the target file.
'overwrite'	Text keyword "OVERWRITE" or "DONT_OVERWRITE". Default value is "DONT_OVERWRITE".
Value:	Integer value, the status code of the operation.

## 10.17.24 FILE\_MANAGER("MOVE", file, target)

Moves a file to another directory.

'file'	Absolute file tag of the file to be moved.
'target'	Absolute directory tag of the directory to become the new parent directory.
Value:	List value with following attributes:
	STATUS Integer value, the status code of the operation.
	NEW_TAG New tag for the moved file (if STATUS = 0)

## 10.17.25 FILE\_MANAGER("RENAME", file, name)

Renames a file.

'file'	Absolute file tag of the file to be renamed.
'name'	Text value, the new file name.
Value:	List value with following attributes:
	STATUS Integer value, the status code of the operation.
	NEW_TAG New tag for the renamed file (if STATUS = 0).

## 10.17.26 FILE\_MANAGER("GET\_ATTRIBUTES", file)

Returns attribute information from one or more files.

'file'	Absolute file tag or vector of file tags of interest.
Value:	List value containing the following attributes:
	STATUS Integer or integer vector value, the status of the query.
	FAILURES Integer value telling the number of failed queries.
	READ_ONLY Boolean or boolean vector value.
	SYSTEM Boolean or boolean vector value, TRUE if exclusively used by the OS.
	HIDDEN Boolean or boolean vector value, TRUE if a 'hidden' file.
	ARCHIVE Boolean or boolean vector value, TRUE if 'archive' attribute is set in the file.
	COMPRESSED Boolean or boolean vector value, TRUE if a compressed file.
	ENCRYPTED Boolean or boolean vector value, TRUE if an encrypted file.
	CREATED_S Time or time vector value, the seconds of the creation.time of the file.
	CREATED_US Integer or integer vector value, the microseconds of the creation time of the file.
	MODIFIED_S Time or time vector value, the seconds of the modification time of the file.

Table continues on next page

MODIFIED_US	Integer or integer vector value, the microseconds of the modification time of the file.
SIZE_KB	Integer or integer vector value, the kilobytes of the size of the file.
SIZE_B	Integer or integer vector value 0 ... 1023, the bytes of the size of the file.

## 10.17.27 Auxiliary functions

### 10.17.28 FM\_APPLICATION\_DIRECTORY[(path)]

Creates a directory tag out of an application relative directory path.

'path'	Text or text vector value, the directory path(s) given in SCIL file name format, e.g. "PAR/DEFAULT".
Value:	The absolute directory tag of the directory specified by 'path' or a vector of such tags.
If argument 'path' is omitted, the tag of the application root directory is returned.	

### 10.17.29 FM\_APPLICATION\_FILE(path)

Creates a file tag out of an application relative file path.

'path'	Text or text vector value, the file path(s) given in SCIL file name format, e.g. "PAR/DEFAULT/MYTOOL.INI".
Value:	The absolute file tag of the file specified by 'path' or a vector of such tags.

### 10.17.30 FM\_COMBINE(tag1 [,tagi]\*, tagn)

Combines two or more drive, directory or file tags to create a new directory or file tag.

'tag1'	A drive or a directory tag (either absolute or relative).
'tagi'	One or more (up to 30) directory tags.
	The tags must be relative directory tags, except for the first one which can be an absolute directory tag using the default drive if 'tag1' is a drive tag.
'tagn'	A directory or file tag or vector of tags.
Value:	A directory or file tag denoting the path from 'tag1' to 'tagn' or a vector of such tags.

The tags given as the arguments must follow the order of file hierarchy. A drive must be first argument if any. Any absolute path containing a drive may not be specified if the first argument is a drive tag. Only one absolute path may be given. An absolute path may not follow a relative path in the argument list. A file path must be the last argument if any.

### 10.17.31 FM\_COMBINE\_NAME(name, extension)

Combines a proper file name and an extension to a file name.

'name'	Text or text vector value, the proper file name(s).
'extension'	Text or text vector value, the file name extension(s).
Value:	Text or text vector value, the combined file name(s).

Either 'name' or 'extension' or both may be vectors. If both, they must be of equal length.

Trailing blanks, if any, are removed from 'name' and 'extension'. If the extension is empty, the name is returned. Otherwise, a period is inserted between name and extension. The resulting file name is not syntax checked.

### 10.17.32 FM\_DIRECTORY(path [,check])

Creates a directory tag out of a directory path or checks a directory path.

'path'	Text or text vector value, directory path(s). Any valid directory path.
'check'	Text keyword "CHECK".
Value:	If "CHECK", a text or text vector value: "ABSOLUTE"            'path' denotes a valid absolute path. "RELATIVE"            'path' denotes a valid relative path. "ERROR"              'path' is not a valid path. Otherwise, an absolute or relative directory tag (a byte string value) or a vector of tags.

### 10.17.33 FM\_DRIVE(name [,check])

Creates a drive tag out of a drive name or checks a drive name.

'name'	Text or text vector value, drive name(s). In Windows, one-letter name optionally followed by ":" or a UNC name.
'check'	Text keyword "CHECK".
Value:	If "CHECK", a text or text vector value "OK" or "ERROR", otherwise a drive tag (a byte string value) or a vector of drive tags.

### 10.17.34 FM\_EXTRACT(tag, component)

Extracts a component from one or more directory or file tags.

'tag'	Any directory or file tag or a vector of such tags.
'component'	Text keyword specifying the component to be extracted.
"DRIVE"	The drive, zero length byte string is returned if 'tag' is not an absolute path.
"PATH"	The intermediate directory path to the last component.
"DIRECTORY"	The directory containing the 'tag' ("DRIVE" and "PATH" combined).
"LAST"	The directory or file denoted by 'tag'.
"APPLICATION_RELATIVE"	The tag relative to the current application directory.
Value:	A byte string or a byte string vector containing the tag(s) of the selected component.

For a missing component, a zero length byte string is returned.

If the selected component is "LAST", a relative directory or file tag is returned.

If the selected component is "PATH" or "DIRECTORY", the result is absolute if 'tag' is absolute and relative if 'tag' is relative.

### **10.17.35 FM\_FILE(path [,check])**

Creates a file tag out of a file path or checks a file path.

'path'	Text or text vector value, file path(s). Any valid file path.
'check'	Text keyword "CHECK".
Value:	If "CHECK", a text or text vector value:
"ABSOLUTE"	'path' denotes a valid absolute path.
"RELATIVE"	'path' denotes a valid relative path.
"ERROR"	'path' is not a valid path.
	Otherwise an absolute or relative file tag (a byte string value) or a vector of tags.

### **10.17.36 FM\_REPRESENT(tag [,option]\*)**

Converts one or more drive, directory or file tags into an OS dependent text representation.

'tag'	Byte string or a byte string vector value: drive, directory or file tag(s).
'option'	One or more of following text keywords in any order
"DRIVE_POSTFIX"	In Windows, append a ":" to a drive name (if a drive tag).
"DIRECTORY_POSTFIX"	In Windows, append a "\ " to directory path (if a directory tag).
"UPPER_CASE"	Use upper case letters.
"LOWER_CASE"	Use lower case letters.
"CAPITALIZE"	Capitalize the names.
Value:	Text or text vector value, the text representation of the given tag(s).

By default, in Windows a drive name is given without the trailing ":", a directory name is given without the trailing "\" and the case used when the tag was created is not changed.

If 'tag' denotes the default drive, an empty string is returned.

#### **10.17.37 FM\_SCIL\_DIRECTORY(name [,check])**

**Creates a directory tag out of a SCIL directory name or checks a SCIL directory name.**

'name'	Text or text vector value, SCIL name(s).
'check'	Text keyword "CHECK".
Value:	If "CHECK", a text or text vector value:  "ABSOLUTE"      'name' denotes a valid absolute directory. "RELATIVE"      'name' denotes a valid application directory. "ERROR"      'name' is not a valid SCIL name.  Otherwise an absolute directory tag (a byte string value) or a vector of tags.

SCIL directory names begin with a slash and are relative to the root directory of the SYS600 installation (usually "\SC"). To create a directory tag for a logical path, use function PATH first to find out the directory name(s).

### **10.17.38 FM\_SCIL\_FILE(name [,option] [,option])**

Creates a file tag out of a SCIL file name or checks a SCIL file name.

'name'	Text or text vector value, SCIL name(s).
'option'	Text keyword "CHECK" or "IGNORE_EXISTING".
Value:	If "CHECK", a text or text vector value:
	"ABSOLUTE"      'name' denotes a valid absolute file.
	"RELATIVE"      'name' denotes a valid application file.
	"ERROR"          'name' is not a valid SCIL name.
	Otherwise an absolute file tag (a byte string value) or a vector of tags.

If "IGNORE\_EXISTING" option is not specified, the function does a lookup for the name given, and returns a tag for the found file. If the file does not exist, a tag for a file that would be created by 'name' is returned. If the option is specified, the lookup is bypassed.

### **10.17.39 FM\_SCIL\_REPRESENT(tag [,case])**

Converts one or more directory or file tags into a SCIL name text representation.

'tag'	Byte string or a byte string vector value: directory or file tag(s).
'case'	One of the following text keywords:
	"UPPER_CASE"      Use upper case letters.
	"LOWER_CASE"      Use lower case letters.
	"CAPITALIZE"      Capitalize the names.
Value:	Text or text vector value, the SCIL name representation of the given tag(s).

A zero length text is returned if the 'tag' cannot be converted into a SCIL name, i.e. it contains a drive or is not located below the root of SCIL file hierarchy (in Windows, "\SC\" by default).

### **10.17.40 FM\_SPLIT\_NAME(file)**

Extracts the proper name and the extension from one or more file names.

'file'	Text or text vector value, file name(s).
Value:	List value with following attributes.
	NAME            Text or text vector value, the proper name(s).
	EXTENSION      Text or text vector value, the extension(s).

## **10.18 Communication functions**

### **10.18.1 SPACOM(message)**

Communicates with a SPACOM unit connected to a COM port.

The SPACOM function sends the 'message' string to the SPACOM unit and returns the reply character string. The function can be used only for communication with SPACOM units connected to the base system. This assumes that the base system attributes SYS:BSD and SYS:BSP have been set (see the System Objects manual).

'message'	Text. The message to be sent to the SPACOM unit.
Value:	Text. Reply string. "N" is returned if the unit replies with NAK, and "T" is returned if the unit does not reply or the reply cannot be interpreted (time-out).

## 10.18.2 TIMEOUT(milliseconds)

Changes the communication time-out.

Changes locally the base system time-out used in the communication with other nodes. The function affects only the context where it is used. The global time-out is specified by the SYS:BTI attribute.

'milliseconds'	Integer expression, >= 0. The time-out in milliseconds.
	0 means that SYS:BTI will be used for time-out.
Value:	The previous time-out value.

The time-out is applied to the following communication:

- Communication with an external application (APL-APL communication).
- Communication with a NET via system (S) objects.
- Communication with a NET via process objects (#SET and #GET command).

Example:

```
#LOCAL OLD, TEMP
OLD = TIMEOUT(10000)           ;The time-out is changed
#SET STA:S... .....          ;Communication with NET
TEMP = TIMEOUT(OLD)           ;The time-out is reset to its previous
                             value.
```

## 10.19 CSV (Comma Separated Value) functions

The CSV functions are used to export data from SYS600 to any Windows application that uses CSV (Comma Separated Value) file format and to import CSV data into a SYS600 application. The spread-sheet program Excel is a good example of such a Windows application.

The CSV format exists as two variants:

- The 'original' variant uses a comma as the field separator character (as the format name states) and a dot as the decimal point character.
- The 'European' variant uses a semicolon as the field separator, because the comma is reserved for the decimal point use.

Both of these variants are supported by the CSV functions.

All the SCIL data types are supported by the CSV functions. For the structured data types (vectors and lists), two representation options are implemented:

- Horizontal: The components are presented on one line, separated by commas, as the output of the DUMP function.
- Vertical: Each component is shown in its own line. Note, that these lines still make up only one field of the CSV format.

The primary target in the design of the CSV functions has been the database export/import functionality. Typically, a CSV record contains the configuration of a SYS600 database object and a CSV field contains the value of one attribute of the object. However, the functions can be used to export and import any data between SCIL and an external application.

## 10.19.1 CSV\_TO\_SCIL(csv, start, field\_info [,option])

Converts a CSV file format record into SCIL data.

'csv'	Text vector, the contents of a CSV file (or a portion of it)	
'start'	Positive integer, the row number in 'csv' to start with.	
'field_info'	A list value describing the contents of the CSV fields or a vector of such list values, if the fields are different. The list contains the following attributes:	
VALUE_TYPE		Text keyword, any value type returned by the OBJECT_ATTRIBUTE_INFO function. This attribute defines the SCIL value type of the field data, or the value type of the elements of a vector value.
VECTOR_LENGTH		Integer, the maximum length of vector data. If the field does not contain a vector, this attribute should be omitted.
ELEMENT_TYPE		Text keyword, see VALUE_TYPE. The value type of the elements of vector data. If the field does not contain a vector, this attribute should be omitted.
'option'	Optional text keyword defining which variant of the CSV format is used:	
"DECIMAL_POINT_IS_COMMAS"	The 'European' variant is assumed.	
"DECIMAL_POINT_IS_UNKNOWN"	The function tries to deduce the variant by itself.	
when omitted	The 'original' variant is assumed.	
Value:	A list with the following attributes:	
STATUS	SCIL status code for the operation, 0 = OK. The next attributes are returned only if STATUS = 0. If STATUS = 678 (SCIL_CVS_CLOSING_QUOTE_MISSING), it may indicate that more data should be read from the CSV file to complete the CSV record.	
SCIL	Vector containing the values of the fields.	
ERROR_FIELDS	Integer vector containing the numbers of the fields (indices of the vector SCIL) that could not be converted into SCIL data. The data type of these elements is set to "NONE".	
ERROR_CODES	Integer vector of the same length as ERROR_FIELDS, SCIL status codes for the failed fields.	
DECIMAL_POINT	One-character text containing the decimal point character (dot or comma). This attribute is returned only when the option "DECIMAL_POINT_IS_UNKNOWN" is given.	
START	The row number in the CSV vector to be used as the argument 'start' in the next call of this function. If START = 0, the whole vector has been converted.	

See function SCIL\_TO\_CSV for the reverse operation.

## 10.19.2 SCIL\_TO\_CSV(data [,option]\*)

Converts SCIL data into a CSV file format record.

'data'	Vector value, the SCIL data to be converted	
'option'	Up to 4 text keywords in any order:	
	"DECIMAL_POINT_IS_COMMA"	The comma is used as the decimal point character and semicolon as the field separator.
	"VERTICAL_TEXT_VECTORS"	Text vectors are shown vertically.
	"VERTICAL_VECTORS"	Vectors, other than text vectors, are shown vertically.
	"VERTICAL_LISTS"	Lists are shown vertically.
Value:	A list with the following attributes:	
	CSV	Text vector containing the CSV record.
	ERROR_FIELDS	Integer vector containing the numbers of the fields (indices of the 'data' vector) that could not be converted into CSV. These fields are empty in the result.
	ERROR_CODES	Integer vector of the same length as ERROR_FIELDS, SCIL status codes for the failed fields.

The default options are the following:

- The decimal point is a dot, the field separator is a comma.
- Text vectors are shown horizontally.
- Other vectors are shown horizontally.
- Lists are shown horizontally

See the function CSV\_TO\_SCIL for the reverse operation.



Some CSV applications (including Excel) have various size restrictions on the CSV data. Therefore, they may fail to import a CSV file generated by this function, if the file is very big (many rows and/or columns) or a field is very long or contains many lines.

## 10.20 DDE client functions

The DDE client functions allow the SYS600 user (application) access to external applications using the DDE (Dynamic Data Exchange) protocol. The DDE client functions establish a DDE link between SYS600 and other Windows applications, such as Microsoft Excel or Word. The SYS600 application works as a client and the other application as a server. The other application must be running when the connection is opened.

Since Windows Vista and Windows Server 2008, the client and server application must be running in the same computer.

The DDE protocol is supported by most Microsoft Windows applications. To use the SYS600 DDE client functions, the user should be familiar with the server application and its DDE functions.

The DDE client functions support the following data transaction commands directed from the client (SYS600) to the server application:

<b>Request</b>	Requests a data transfer from a server to a client.
<b>Poke</b>	Writes data in the server application.
<b>Execute</b>	Executes an item (a command) in the server application (if execute supported by the application).

The DDE client functions use the following three identifiers to address data in the server application:

<b>Service name</b>	Usually the name of the server application.
<b>Topic name</b>	Identifies a logical data context, e.g. a file.
<b>Item name</b>	Identifies the data.

The DDE functions in SYS600 use status codes listed below, most of which are DDE protocol status codes. The status code texts can be, and should be, used as such in SCIL expressions. They are defined as predefined integers in the SYS600 kernel.

STATUS codes	Decimal value
DMLERR_NO_ERROR	0
DMLERR_ADVACKTIMEOUT	16384
DMLERR_BUSY	16385
DMLERR_DATAACKTIMEOUT	16386
DMLERR_DLL_NOT_INITIALIZED	16387
DMLERR_DLL_USAGE	16388
DMLERR_EXECACKTIMEOUT	16389
DMLERR_INVALID_PARAMETER	16390
DMLERR_LOW_MEMORY	16391
DMLERR_MEMORY_ERROR	16392
DMLERR_NOTPROCESSED	16393
DMLERR_NO_CONV_ESTABLISHED	16394
DMLERR_POKEACKTIMEOUT	16395
DMLERR_POSTMSG_FAILED	16396
DMLERR_REENTRANCY	16397
DMLERR_SERVER_DIED	16398
DMLERR_SYS_ERROR	16399
DMLERR_UNADVACKTIMEOUT	16400
DMLERR_UNFOUND_QUEUE_ID	16401
DMLERR_ITEM_TOO_LONG	20483

## 10.20.1 DDE\_CONNECT(service, topic)

Opens a connection to an external application.

'service'	Text. The service name the remote application responds to. Usually the same as the application name.	
'topic'	Text. A valid topic name within the remote application.	
Value:	A list containing the following two attributes:	
	STATUS	A predefined integer (see above). The value indicates whether the function was successfully executed or not. DMLERR_NO_ERROR (= 0) means that the operation succeeded.
	CONNECTION_ID	Integer, 1 ... 10. The id-number of the connection. If the connection did not succeed, the CONNECTION_ID is undefined.

The connection gets an identification number which is used in the subsequent DDE operations. The DDE connections are SCIL context specific which means that they are automatically closed when a picture is exited, when a command procedure has been executed to the end or when the dialog system is deleted. Each SCIL context can have up to 10 open DDE connections simultaneously.

The remote application must be running when the DDE\_CONNECT function is issued. Likewise, the topic for which DDE\_CONNECT is issued must be available. For instance, for Excel this means that the desired spreadsheet must be opened. The remote application can be started from SCIL with the OPS\_CALL command. At the same time the desired topic (e.g. a file) can be opened. A topic can also be opened later by sending an open command to the server application using the DDE\_EXECUTE function.

Example:

Connecting to Excel 5.0, sheet 1:

```
#LOCAL RESULT, CONN
RESULT = DDE_CONNECT("EXCEL", "SHEET1")
#IF RESULT.STATUS == DMLERR_NO_ERROR #THEN #BLOCK
    CONN = RESULT.CONNECTION_ID
#BLOCK_END ;Connection was successful
#ELSE #BLOCK ;Connection not successful
#BLOCK_END
```

## 10.20.2 DDE\_DISCONNECT(connection\_id)

Closes the DDE connection.

'connection_id'	Integer, 1 ... 10. The DDE connection identifier obtained using the DDE_CONNECT.	
Value:	A list containing the following attribute:	
	STATUS	A predefined integer, see above. The value indicates whether the function was successfully executed or not. DMLERR_NO_ERROR (= 0) means that the operation succeeded.

Example:

Disconnecting from a remote application (e.g. Excel):

```
RESULT = DDE_DISCONNECT(CONN )
#IF RESULT.STATUS == DMLERR_NO_ERROR #THEN #BLOCK
;Connection was successfully closed.
#BLOCK_END
```

```
#ELSE
;Connection not successfully closed.
```

### 10.20.3 DDE\_REQUEST(connection\_id, item [,timeout])

Requests data from a remote application.

A connection to the application must be open (see above).

'connection_id'	Integer, 1 ... 10. The DDE identifier obtained using the DDE_CONNECT.
'item'	Text. A valid DDE item in the remote application.
'timeout'	Integer, 0 ... 1000. The DDE transaction timeout in seconds. The parameter is optional. Default value = 20 s.
Value:	A list containing the following two attributes:
	STATUS      A predefined integer (see above). The value indicates whether the function was successfully executed. DMLERR_NO_ERROR (= 0) means that the operation succeeded.
	DATA        Text (max length 65 535), the requested data. If STATUS is not equal to DMLERR_NO_ERROR, the DATA is undefined.

Example:

Requesting data from Excel:

```
#LOCAL RESULT, ITEM_VALUE
RESULT = DDE_REQUEST(CONN, "R1C1", 30)
#IF RESULT.STATUS == DMLERR_NO_ERROR #THEN #BLOCK
    ITEM_VALUE = RESULT.DATA
    ;The request was successful
#BLOCK_END
#ELSE #BLOCK
    ;The request was NOT successful
#BLOCK_END
```



The cell references in Excel are language dependent if the reference style R1C1 is used (R for row and C for column). To check the type of reference style and the cell reference notation of the Excel version in use, select: **Tools > Options > General tab**.

### 10.20.4 DDE\_POKE(connection\_id, item, value [,timeout])

Sets the value of 'item' in a remote application.

A connection to the application must be open (see above).

'connection_id'	Integer, 1 ... 10. The DDE identifier obtained using the DDE_CONNECT.
'item'	Text. A valid DDE item in the remote application.
'value'	Text. The value which will be set into item.
'timeout'	Integer, 0 ... 1000. The DDE transaction time-out in seconds. The parameter is optional. Default value = 20 s.
Value:	A list containing the following attribute:
	STATUS      A predefined integer (see above). The value indicates whether the function was successfully executed. DMLERR_NO_ERROR (= 0) means that the operation succeeded.

Example:

Setting "Time of Day" in cell "R1C1" in Excel:

```
#LOCAL RESULT, CONN
...
RESULT = DDE_POKE(CONN, "R1C1", TOD, 30)
#IF RESULT.STATUS == DMLERR_NO_ERROR #THEN #BLOCK
    ;The value was successfully set.
#BLOCK_END
#ELSE #BLOCK
    ;The value was NOT successfully set.
#BLOCK_END
```

## 10.20.5 DDE\_EXECUTE(connection\_id, statement [,timeout])

Executes a statement in a remote application.

The connection to the application must be open. To be able to use this function, the DDE documentation of the server application is needed.

'connection_id'	Integer, 1 ... 10. The DDE connection identifier obtained using the DDE_CONNECT.
'statement'	Text. A valid executable statement in the remote application (see the documentation of the application in question).
'timeout'	Integer, 0 ... 1000. The DDE transaction time-out in seconds. The parameter is optional. Default value = 20 s.
Value:	A list containing the following attribute:
STATUS	A predefined integer (see above). The value indicates whether the function was successfully executed. DMLERR_NO_ERROR (= 0) means that the operation succeeded.

Example:

Executing a statement in Excel:

```
#LOCAL RESULT, CONN
...
RESULT = DDE_EXECUTE(CONN, "%o~", 30)
#IF RESULT.STATUS== DMLERR_NO_ERROR #THEN #BLOCK
    ;The request command was successfully executed
#BLOCK_END
#ELSE #BLOCK
    ;The request was NOT successful.
#BLOCK_END
```

The string "%o~" corresponds to <Alt+o><ENTER> in Excel and opens a dialog for cell formatting.



Use the DDE\_EXECUTE function with caution! Different applications respond differently to the DDE\_EXECUTE statement. Excel, for example, must be the active application on the desktop to be able to execute a submitted DDE\_EXECUTE command.

## 10.21 DDE server functions

SYS600 applications can be accessed from external Windows applications using the DDE protocol so that the SYS600 application works as a server and the other application as a client.

DDE Server supports the following SCIL data types:

Integer	
Text	
Time	Time will be returned as seconds count from 1.1.1978.
Boolean	Boolean will be returned as 0 or 1.
Real	Real will be returned to client application as SCIL real format ("." as decimal separator)

The vector data type is not supported directly, but can be used with the help of the DDE\_VECTOR function described below. SYS600 real data can be transformed into text data with user defined decimal separator using the DDE\_REAL function (see below). The data types Bit string, Byte string and List are not supported.

### 10.21.1 DDE\_VECTOR(vector, decimal\_separator, list\_separator)

Creates a DDE style list with a user defined list separator.

The function transforms a SCIL vector into a format readable in the client in the application.

'vector'	Vector. A valid SCIL expression. The vector may only contain real, integer, text and boolean elements. Note that the vector will be transformed into a text which can contain no more than 65 535 characters.
'decimal_separator'	Text. A valid separator in the client application.
'list_separator'	Text. A valid separator in client application.
Value:	A text composed of the elements in 'vector' separated by the list separator (see the example below).

If a vector element is of invalid data type or missing, there is an empty value in result string. In other words, there are two consecutive list separator characters in the result string.

Example:

```
#LOCAL A = DDE_VECTOR(VECTOR("TEST",1,2.0),".",";")
;Result: A == "TEST;1;2.0"
```

### 10.21.2 DDE\_REAL(real, separator)

Creates a DDE style real number with a user defined decimal separator.

'real'	Real. A valid SCIL expression.
'separator'	Text. A valid decimal separator in client application.
Value:	A text value with defined decimal separator.

Example:

```
@A = DDE_REAL(1.23,"")
;Result: %A == "1,23"
```

## 10.22 ODBC functions

The ODBC functions provide means for accessing databases and applications that support SQL (Structured Query Language), e.g. databases built with Microsoft Access, Paradox and Oracle. SYS600 uses the database interface Microsoft ODBC (Open Database Connectivity) and SQL statements.

Access to databases requires that an ODBC driver for the database management system in question has been installed and configured in the SYS600 computer where the ODBC functions are executed. For more information, see the ODBC documentation.

To be able to use the ODBC functions, the programmer should be familiar with SQL statements. Refer to suitable SQL manuals and books.

The ODBC functions may return the following seven status codes, which can be used as such in SCIL expressions because they are defined as predefined integers:

0	SQL_SUCCESS
1	SQL_SUCCESS_WITH_INFO
-1	SQL_ERROR
100	SQL_NO_DATA_FOUND
2	SQL_STILL_EXECUTING
99	SQL_NEED_DATA
2001	SQL_UNSUPPORTED_DATATYPE

In certain cases, also an error code may be returned (for status codes SQL\_ERROR and SQL\_SUCCESS\_WITH\_INFO). The error codes are the ODBC error codes listed in appendix A.

### 10.22.1 SQL\_CONNECT(source, user, password)

Opens an ODBC connection to a data source.

The data source may be situated in the same computer or in another computer on the network. The connection gets an identification number which is used in the subsequent ODBC functions.

The ODBC connections are SCIL context specific. This means that when the context, e.g. the picture or command procedure, is exited, the connections are automatically closed.

Up to 10 ODBC connections may be open simultaneously.

'source'	Text. The data source name as defined by the ODBC administration tool.
'user'	Text. A text string containing the user name that is to be used when accessing the data source specified in 'source'. An empty string if no user name needed.
'password'	Text. A text string containing the password for 'user' when accessing the data source specified in 'source'. An empty string if no password needed.
Value:	A list containing the following three attributes:

Table continues on next page

STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
CONNECTION_ID	Integer, 1 ... 10. The identification number of the connection. Returned only when the operation did succeed.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

**Example:**

Opening the connection to a data source named ACCESS:

```
@RESULT = SQL_CONNECT("ACCESS", "MICRO", "SCADA")
#IF RESULT:VSTATUS == SQL_SUCCESS OR-
    RESULT:VSTATUS == SQL_SUCCESS_WITH_INFO #THEN #BLOCK
@CONN = RESULT:VCONNECTION_ID
;Continue the database
interaction.
#BLOCK_END
#ELSE #BLOCK
;The connection was not successful.
#BLOCK_END
```

## 10.22.2 SQL\_DISCONNECT(connection\_id)

Closes the ODBC connection defined by the argument.

If SQL\_BEGIN\_TRANSACTION has been issued, SQL\_COMMIT is run automatically before the connection is closed (see below). All statements associated with the connection are freed.

'connection_id'	Integer 1 ... 10. The connection identification of the connection that is to be disconnected.
Value:	A list containing the following two attributes:
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

**Example:**

Closing the connection opened in the example above:

```
@RESULT = SQL_DISCONNECT(%CONN)
#IF RESULT:VSTATUS <> SQL_SUCCESS AND-
    RESULT:VSTATUS <> SQL_SUCCESS_WITH_INFO #THEN #BLOCK
;The connection was not successfully disconnected.
#BLOCK_END
```

## 10.22.3 SQL\_EXECUTE(connection\_id, SQLstring [,timeout])

Executes an SQL statement.

'connection_id'	Integer 1 ... 10. The identification number of the connection on which the statement is to be executed.	
'SQLstring'	Text or text vector. The SQL statement that is to be executed. For more information, see SQL manuals.	
'timeout'	Non-negative integer. The maximum time (in seconds) the result is waited for. Zero value specifies no timeout. The default value is 20 seconds.	
Value:	A list containing the following three attributes:	
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.	
STATEMENT_ID	Integer, 1 ... 100. The identification number of the statement. Returned only if the operation did succeed.	
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.	

The function executes the SQL statement on the connection specified by the 'connection\_id' argument. It reserves a statement identification number which is used in a possible subsequent SQL\_FETCH function. Each connection may have up to ten simultaneous execute statements.

Example:

Deleting all records in the table "MYTABLE" where the value in the column "AGE" is equal to 18:

```
@SQLSTMT = "DELETE FROM MYTABLE WHERE AGE = 18"
@RESULT = SQL_EXECUTE(%CONN, %SQLSTMT)
#IF RESULT:VSTATUS == SQL_SUCCESS OR-
RESULT:VSTATUS == SQL_SUCCESS_WITH_INFO -
#THEN #BLOCK
    @STMT = RESULT:VSTATEMENT_ID ;Continue the database interaction.
#BLOCK_END
#ELSE #BLOCK ;The statement was not successfully executed.
#BLOCK_END
```

See also the SQL\_FETCH example below.

#### 10.22.4 SQL\_FETCH(statement\_id)

Fetches a row of data from a result set obtained by SQL\_EXECUTE.

'statement_id'	Integer 1 ... 100. The statement identification number returned by SQL_EXECUTE.	
Value:	A list containing the following three attributes:	
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.	
DATA	A vector containing data fetched from a result table. Returned only if the operation did succeed.	
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.	

The function has no meaning if SQL\_EXECUTE not executed.

Example:

Reading names and addresses from a database:

```
@SQLSTMT = "SELECT NAME, ADDRESS FROM EMPLOYEE"
@RESULT = SQL_EXECUTE(%CONN, %SQLSTMT)
```

```

#IF RESULT:VSTATUS == SQL_SUCCESS OR-
RESULT:VSTATUS == SQL_SUCCESS_WITH_INFO -
#THEN #BLOCK
@STMT = RESULT:VSTATEMENT_ID
#LOOP
@RESULT=SQL_FETCH(%STMT)
#IF RESULT:VSTATUS == SQL_SUCCESS OR-
RESULT:VSTATUS == SQL_SUCCESS_WITH_INFO -
#THEN #BLOCK
..... ; FETCH succeeded
#BLOCK_END
#ELSE #LOOP_EXIT
#LOOP_END
#BLOCK_END
;RESULT:VDATA(1) and RESULT:VDATA(2) contain
;the name and address respectively.

```



The UNICODE character set is not supported by the function. Therefore, any UNICODE data to be fetched should be converted to one-byte characters using 'convert' function in the SELECT clause.

## 10.22.5 SQL\_FREE\_STATEMENT(statement\_id)

Frees the specified statement and stops processing associated with the statement.

'statement_id'	Integer, 1 ... 100. The statement identification returned from SQL_EXECUTE.
Value:	A list containing the following two attributes:
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

This function must be used if several calls to SQL\_EXECUTE are issued on the same connection. All open statements associated with the connection will be freed automatically when a connection is closed.

Example:

```

@RESULT=SQL_FREE_STATEMENT (%STMT)
#IF RESULT:VSTATUS <> SQL_SUCCESS AND-
    RESULT:VSTATUS <> SQL_SUCCESS_WITH_INFO -
#THEN #BLOCK
    ;The statement was not successfully freed.
#BLOCK_END

```

## 10.22.6 SQL\_BEGIN\_TRANSACTION(connection\_id)

Marks the beginning of a transaction.

'connection_id'	Integer, 1 ... 10. The identification of the connection on which the transaction is to be executed.
Value:	A list containing the following two attributes:
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

The transaction will be completed when SQL\_COMMIT or SQL\_ROLLBACK is encountered (see below). Disconnecting the connection will also commit the transaction.

Example:

```
@RESULT = SQL_BEGIN_TRANSACTION(%CONN)
#IF RESULT:VSTATUS == SQL_SUCCESS OR-
    RESULT:VSTATUS == SQL_SUCCESS_WITH_INFO -
        #THEN #BLOCK
        @SQLSTMT = "DELETE FROM MYTABLE WHERE AGE = 18"
        @RESULT1 = SQL_EXECUTE(%CONN, %SQLSTMT)
        @SQLSTMT = "UPDATE MYTABLE SET AGE = 18 WHERE AGE = 17"
        @RESULT2 = SQL_EXECUTE(%CONN, %SQLSTMT)
#IF (RESULT1:VSTATUS == SQL_SUCCESS OR-
    RESULT1:VSTATUS == SQL_SUCCESS_WITH_INFO) AND-
        (RESULT2:VSTATUS == SQL_SUCCESS OR-
            RESULT2:VSTATUS == SQL_SUCCESS_WITH_INFO) -
                #THEN #BLOCK
                @RESULT = SQL_COMMIT(%CONN)
                #IF RESULT:VSTATUS <> SQL_SUCCESS AND-
                    RESULT:VSTATUS <> SQL_SUCCESS_WITH_INFO - #THEN #BLOCK
                    ;The commit was not successful
                #BLOCK_END
                #BLOCK_END
#ELSE #BLOCK
                @RESULT = SQL_ROLLBACK(%CONN)
                #IF RESULT:VSTATUS <> SQL_SUCCESS AND-
                    RESULT:VSTATUS <> SQL_SUCCESS_WITH_INFO - #THEN #BLOCK
                    ;The rollback was not successful
                #BLOCK_END
            #BLOCK_END
#BLOCK_END
#BLOCK_END
#ELSE #BLOCK
    ;The transaction could not be started.
#BLOCK_END
```

## 10.22.7 SQL\_COMMIT(connection\_id)

Commits a transaction the start of which was marked with SQL\_BEGIN\_TRANSACTION.

'connection_id'	Integer, 1 ... 10. The identification number of the connection on which a transaction is to be committed.
Value:	A list containing the following two attributes:
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

Example:

See the example related to the SQL\_BEGIN\_TRANSACTION function above.

## 10.22.8 SQL\_ROLLBACK(connection\_id)

Rolls back a transaction started with SQL\_BEGIN\_TRANSACTION.

'connection_id'	Integer, 1 ... 10. The connection identification of the connection on which a transaction is to be rolled back.
Value:	A list containing the following two attributes:
STATUS	A predefined integer (see above). The values SQL_SUCCESS and SQL_SUCCESS_WITH_INFO indicate a successful execution.
ERROR_CODE	Text. A five-character ODBC error code. Returned only when STATUS is SQL_ERROR or SQL_SUCCESS_WITH_INFO.

Example:

See the example related to the SQL\_BEGIN\_TRANSACTION function above.

## 10.23 OPC Name Database functions

OPC Name Database functions are used to maintain the OPC Name Database. The database is used to give OPC style (alias) names for SYS600 objects and their attributes.

An OPC name (item id) is a hierarchical name consisting of any number of fields separated by dots:

"field1.field2. ... .fieldn"

Each field may contain any visible characters, except for colons and dots. The name is case-sensitive. Single embedded spaces are allowed in the field, leading and trailing spaces are not accepted. The maximum length of the name is 255 characters.

Examples of valid OPC names:

- "South Tipperary.Kilkenny.Relay1.Breaker.Position"
- "Äänekosken ala-asema.Laukaan syöttö.Katkaisija.Tila"

The names defined in the database and the values of ON attributes of the application objects share the same name space. They must be unique within an application. Consequently, the user cannot, for example, define an OPC name that is identical to the value of the ON attribute of some process object. See the Application Objects manual for the description of the ON attribute.

Each OPC name is mapped to a SYS600 object or an object attribute. Any of the object types P, X, H, F, D, C, T, A, G, B and S, or any of their attributes may be used. It is even allowed to give an OPC name for an element (or a slice) of a vector valued attribute. More than one OPC name may refer to the same SYS600 object.

Examples of valid references given in the SCIL object notation:

ABC:P5	The process object ABC, index 5
DEF:DOS	The OS (Object Status) attribute of the data object DEF
SYS:BPA	The PA (Primary Application) of the base system
APL:BUV(1..5)	The five first user variable elements of the application

External references are not supported. The following is not a valid reference:

ABC:3P5	A process object in application 3 -- not valid
---------	--

The OPC Name Database is stored in the application file APL\_OPCNAM.SDB. It is implemented as a SCIL database for fast and easy access and flexibility. The OPC names act as section

names, the SCIL mapping is given in the data of the section. For SCIL databases, see [Section 6.6](#).

At the application start-up, the complete OPC name space is created by combining the names (ON attribute values) from the process and report database with the ones read from the OPC Name Database.

When an application object is deleted or renamed, the OPC names that refer to it or any of its attributes are also deleted.

The OPC Name Database is maintained in real-time by the SCIL function OPC\_NAME\_MANAGER.

### 10.23.1 OPC\_NAME\_MANAGER(function, apl [,argument]\*)

OPC Name Database maintenance.

'function'	Text keyword, "LIST", "PUT", "GET" or "DELETE".
'apl'	Integer 0 ... 250, the logical application number 0 = current application External applications are supported
'argument'	Additional argument(s) required by the 'function'
Value:	A list always containing the attribute: STATUS SCIL status code (0 = OK) Additional attributes are returned depending on the 'function'.

### 10.23.2 OPC\_NAME\_MANAGER("LIST", apl)

Lists the OPC item names found in the OPC Name Database.

Value:	A list of attributes: STATUS SCIL status code (0 = OK) LIST A text vector containing the names
--------	--

### 10.23.3 OPC\_NAME\_MANAGER("PUT", apl, name, definition)

Creates a new name in the OPC Name Database or overwrites an existing one.

'name'	A text, the OPC name to be added
'definition'	A list containing at least the following attribute: SCIL A text value, the SCIL attribute reference, see above. Additional attributes, such as descriptive texts may be given if wanted. They are stored, but not otherwise processed by the base system.
Value:	A list of one attribute: STATUS SCIL status code (0 = OK)

The given name and the SCIL attribute reference are syntax checked according to the rules above.

### 10.23.4 OPC\_NAME\_MANAGER("GET", apl, name)

Reads the definition of a name from the OPC Name Database.

'name'	A text, the OPC name to be read	
'Value:	A list of attributes:	
	STATUS	SCIL status code (0 = OK)
	DEFINITION	A list, the definition stored by the "PUT" function

## 10.23.5 OPC\_NAME\_MANAGER("DELETE", apl, name)

Deletes a name from the OPC Name Database.

'name'	A text, the OPC name to be deleted	
'Value:	A list of one attribute:	
	STATUS	SCIL status code (0 = OK)

## 10.24 OPC functions

The following SCIL functions are used to search for existing OPC servers in the network, examine their name space, cross-check the SYS600 process database and the server name space, and perform some specific actions on an open OPC client - server connection.

### 10.24.1 OPC\_AE\_ACKNOWLEDGE(apl, ln, ix, ack\_id [, comment [, cookie, active\_time]])

Acknowledges a condition in an OPC A&E server.

'apl'	An integer, the logical application number, 0 = current application. The application must be local.	
'ln'	A text, the logical name of the process object to be acknowledged	
'ix'	An integer, the process object index	
'ack_id'	A text, the acknowledger ID. This value is seen by other clients as the acknowledger of the alarm. If an empty text is given, the MicroSCADA user name is used.	
'comment'	Any text supplied by the acknowledger (optional)	
'cookie'	An integer, an opaque number supplied by the server to identify the event. This is the value of the CK attribute of the process object at the time of alarm event.	
'active_time'	A byte string, the alarm time in a format required by the server. This is the value of the CQ attribute of the process object at the time of alarm event.	
'Value:'	An integer, the SCIL status code of the operation Some possible values:	
0	OK_STATUS	
2	OBSOLETE_STATUS, the alarm has already been acknowledged	
690	SCIL_OPC_CLIENT_NOT_RUNNING, there is no running OPC client for (apl, unit).	
691	SCIL_INVALID_ACK_TIME, the 'cookie' or 'active_time' argument not accepted by the server	
689	SCIL_OPC_COMMUNICATION_ERROR	

The arguments 'cookie' and 'active\_time' are optional, but if one is given the other one must also be given. If they are omitted, the CK and CQ attribute of the process object are used.

More information on acknowledgement of OPC A&E conditions is found in the Application Objects manual.

## 10.24.2 **OPC\_AE\_NAMESPACE(nodenr [, clsid1 [, root]]), OPC\_AE\_NAMESPACE(nodename, clsid2 [, root [, user, password]]])**

Lists the name space of an OPC A&E server.

'nodenr'	An integer 1 ... 250, the node number of the type OPC_AE node object describing the OPC A&E server, or the node number of another type node object locating the computer where the server is found.
'clsid1'	A text, the class ID of the server (optional). This argument is given only when the 'nodenr' argument only locates the computer (node type is not OPC_AE). Otherwise, it is omitted or given as an empty text.
'root'	A text, the OPC item ID of the browsing root (optional) If not given, the entire name space is listed.
'nodename'	A text, the name or IP address of the node where the server is located A null string denotes the current node.
'clsid2'	A text, the class ID of the server. This value is obtained for example by the OPC_AE_SERVERS function.
'user'	A text, the name of the user account under which the remote OPC server is launched (optional). The name may be prefixed by the domain name, for example "DOMAIN\USER".
'password'	A text, the password of the user.
Value:	A list of the following attributes:
	STATUS
	An integer, SCIL status code of the operation The following attributes are returned only if STATUS = 0 (OK)
	EVENT_CATEGORIES
	A list vector, each element describes one event category by an event category descriptor, see below.
	AREA_BROWSING_SUPPORTED
	Boolean TRUE if the OPC A&E server supports area browsing, FALSE if not
	ROOT
	A list value, an area node descriptor containing the entire event source hierarchy below the 'root', see below. This attribute is returned only if AREA_BROWSING_SUPPORTED = TRUE.

An event category descriptor is a data structure that describes one event category in an OPC A&E server. It is a list of the following attributes:

ID	An integer, the numeric ID of the category
DESCRIPTION	A text description of the category
TYPE	A text keyword, which tells the type of the events of the category:
"SIMPLE"	Simple events
"TRACKING"	Tracking events
"CONDITION"	Condition events

Table continues on next page

CONDITIONS	A list vector, present only if TYPE = "CONDITION". Each element describes one of the conditions of the category by the following attributes:	
NAME	NAME	The name of the condition.
SUBCONDITIONS	SUBCONDITIONS	A text vector containing the names of the subconditions of the condition.
 An area node descriptor is a data structure that describes one area in the OPC event source hierarchy tree. It is a list with the following five attributes:		
AREA_NAMES	AREA_NAMES	A text vector containing the names of the area nodes directly below this node.
AREAS	AREAS	A list vector containing the area node descriptors of these descendant areas (recursive definition!).
SOURCE_NAMES	SOURCE_NAMES	A text vector containing the names of OPC event sources directly below this node.
SOURCE_QUALIFIED NAMES	SOURCE_QUALIFIED NAMES	A text vector containing the fully qualified names of these sources.
SOURCE_CONDITIONS	SOURCE_CONDITIONS	A vector of vectors containing the names of the conditions that these sources have. Each element is a text vector that lists the names of the conditions associated to the event source.

In the first form of the function call (i.e. the node is defined by a base system node number), the possible user account and its password are defined by the node object attribute OP (NODn:BOP). See the System Objects manual for the details.

### 10.24.3 **OPC\_AE\_REFRESH(apl, unit)**

Requests a refresh from an external OPC A&E server.

'apl'	An integer, the logical application number, 0 = current application. The application must be local.
'unit'	An integer, the number of the process database unit that runs the OPC A&E client
Value:	An integer, the SCIL status code of the operation (0 = OK)

An OPC A&E refresh causes the server to (re)send an event notification for all active and unacknowledged inactive conditions.

If the A&E server provides a full implementation of the OPC A&E specification, this function is never needed. If not, the function may be used to get initial values for OPC Event type process objects that are created or taken into use while the system is running.

### 10.24.4 **OPC\_AE\_SERVERS(nodenr), OPC\_AE\_SERVERS(nodename [, user, password])**

Lists the OPC A&E servers found in a network node.

'nodenr'	An integer 1 ... 250, the node number of the node object locating the computer to be searched for the servers.
'nodename'	A text, the name or IP address of the node to be searched. A null string denotes the current node.
'user'	A text, the name of the user account under which the remote OPC server browser is launched (optional). The name may be prefixed by the domain name, for example "DOMAIN\USER".
'password'	A text, the password of the user.
Value:	A list with the following attributes:
STATUS	SCIL status code of the operation The following attribute is returned only if STATUS = 0 (OK)
SERVERS	A list vector. Each element describes an A&E server by the following 3 text attributes:
NAME	The name of the server
PROGID	The program ID of the server
CLSID	The class ID of the server

In the first form of the function call (i.e. the node is defined by a base system node number), the possible user account and its password are defined by the node object attribute OP (NODn:BOP). See the System Objects manual for the details.

## 10.24.5 OPC\_AE\_VALIDATE(apl, unit)

Cross-checks a process database and the name space of an OPC A&E server.

'apl'	An integer, the logical application number, 0 = current application
'unit'	An integer, the number of the process database unit that runs the OPC A&E client
Value:	A list of the following attributes:
STATUS	SCIL status code of the operation The following attributes are returned only if STATUS = 0 (OK)
AREA_BROWSING_SUPPORTED	Boolean TRUE if the server implements the IOPCEventAreaBrowser interface, FALSE if not. If the interface is not implemented, the attributes INVALID_SOURCES and UNDEFINED_SOURCES are not present.
SOURCE_CONDITION_QUERY_SUPPORTED	Boolean TRUE if the server implements the QuerySourceConditions method, FALSE if not. If the method is not implemented, the attribute INVALID_SOURCE_CONDITIONS is not present.
INVALID_CATEGORY_IDS	A list vector of categories described in the database but not found in the server. Each element of the vector is a list of the following two attributes:
EH	The name of the erroneous event handling object
CI	The category ID in the database
EVENT_TYPE_MISMATCHES	A vector of event type mismatch descriptors. The descriptor is a list that describes one mismatch by the following four attributes:
EH	The name of the erroneous event handling object
CI	The category ID in the event handling object

Table continues on next page

ET	The event type in the event handling object
CATEGORY_EVENT_TYPE	The event type of the category in the server
<b>INVALID_CONDITIONS</b>	
A vector of conditions described in the database but not found in the server. Each element is a list of the following two attributes:	
EH	The name of the erroneous event handling object
CN	The name of the missing condition
<b>INVALID_SOURCES</b>	
A vector of invalid source name descriptors. Each descriptor reports a used event source name (IN attribute value) that is not found in the server. Present only if AREA_BROWSING_SUPPORTED = TRUE. The invalid source name descriptor is a list of the following two attributes:	
IN	The invalid source name
OBJECTS	The process objects that have this IN attribute value as a vector of lists of two attributes:
LN	The name of the process object
IX	The index of the process object
<b>INVALID_SOURCE_CONDITIONS</b>	
A vector of invalid source condition descriptors. Each descriptor reports a process object that either refers to an invalid condition or to a condition that the event source does not have. Present only if SOURCE_CONDITION_QUERY_SUPPORTED = TRUE. The invalid source condition descriptor is a list of the following five attributes:	
LN	The name of the process object
IX	The index of the process object
IN	The event source (IN attribute)
EH	The event handling object
CN	The name of the invalid condition
<b>UNDEFINED_CONDITIONS</b>	
A text vector containing the names of the conditions that are defined in the server but not in the process database.	
<b>UNDEFINED_SOURCES</b>	
A text vector containing the names of the event sources that are defined in the server but not in the process database. Present only if AREA_BROWSING_SUPPORTED = TRUE.	

## 10.24.6 **OPC\_DA\_NAMESPACE(nodenr [, clsid1 [, root]]), OPC\_DA\_NAMESPACE(nodename, clsid2 [, root [, user, password]]])**

Lists the OPC item hierarchy of an OPC Data Access server.

'nodenr'	An integer 1 ... 250, the node number of the type OPC_DA node object describing the OPC DA server, or the node number of another type node object locating the computer where the server is found.
'clsid1'	A text, the class ID of the server (optional). This argument is given only when the 'nodenr' argument only locates the computer (node type is not OPC_DA). Otherwise, it is omitted or given as an empty text.
'root'	A text, the OPC item ID of the browsing root (optional) If not given, the entire name space is listed.
'nodename'	A text, the name or IP address of the node where the server is located A null string denotes the current node.
'clsid2'	A text, the class ID of the server. This value is obtained for example by the OPC_DA_SERVERS function.
'user'	A text, the name of the user account under which the remote OPC server is launched (optional). The name may be prefixed by the domain name, for example "DOMAIN\USER".
'password'	A text, the password of the user.
Value:	A list with the following three attributes:
STATUS	An integer, SCIL status code of the operation The following attributes are returned only if STATUS = 0 (OK)
SUPPORTED	Boolean TRUE if the OPC DA server supports address space browsing, FALSE if not
ROOT	A list value, a name space node descriptor containing the entire item hierarchy below the 'root', see below. This attribute is returned only if SUPPORTED = TRUE.

A name space node descriptor is a data structure that describes one node in the OPC item hierarchy tree. It is a list with the following six attributes:

BRANCH_NAMES	A text vector containing the names of the branch nodes directly below this node.
BRANCHES	A list vector containing the name space node descriptors of these descendant branch nodes (recursive definition!).
ITEM_NAMES	A text vector containing the names of OPC items directly below this node.
ITEM_IDS	A text vector containing the fully qualified OPC item ID's of these items.
ITEM_DATATYPES	An integer vector containing the native data types of these items. See the OPC and Windows OLE documentation for the VARTYPE encoding of the data types.
ITEM_ACCESS_RIGHTS	An integer vector containing the access rights of these items encoded as follows:
0	No access rights
1	Read only
2	Write only
3	Read / write

In the first form of the function call (i.e. the node is defined by a base system node number), the possible user account and its password are defined by the node object attribute OP (NODn:BOP). See the System Objects manual for the details.

## 10.24.7 OPC\_DA\_REFRESH(apl, unit, group [, wait])

Requests a refresh of an item group from an external OPC DA server.

'apl'	An integer, the logical application number, 0 = current application. The application must be local.
'unit'	An integer, the number of the process database unit that runs the OPC DA client
'group'	A text, the name of the item group
'wait'	A non-negative integer, milliseconds to wait for the reply. Optional, default value is 0 (do not wait for reply)
Value:	An integer, the SCIL status code of the operation: 0 = OK_STATUS 8410 = OPCC_REFRESH_TIMEOUT Other error status codes

OPC\_DA\_REFRESH requests a refresh of an item group from the connected OPC Data Access Server and optionally waits until the item values have been updated in the process database.

Typically, this function is used for item groups whose update rate is set to -1. Such a group is not cyclically updated, see the System Objects manual, attribute NOD:BOP.

The function may be used for cyclically updated groups as well, for example when the update cycle is long and fresh item values are needed in the middle of the cycle.

Example:

```
#LOCAL REFRESH_STATUS
...
REFRESH_STATUS = OPC_DA_REFRESH(0, 100, "PassiveGroup", 2000)
#IF REFRESH_STATUS == 0 #THEN -
    .SET INFO._TITLE = "Refreshed"
#ELSE _ IF REFRESH_STATUS == STATUS_CODE("OPCC_REFRESH_TIMEOUT") #THEN -
    .SET INFO._TITLE = "The server did not reply in 2 seconds"
#ELSE .SET INFO._TITLE = "? Failed: " +
STATUS_CODE_NAME(REFRESH_STATUS)
```

## 10.24.8 OPC\_DA\_SERVERS(nodenr), OPC\_DA\_SERVERS(nodename [, user, password])

Lists the OPC Data Access servers found in a network node.

'nodenr'	An integer 1 ... 250, the node number of the node object locating the computer to be searched for the servers.
'nodename'	A text, the name or IP address of the node to be searched. A null string denotes the current node.
'user'	A text, the name of the user account under which the remote OPC server browser is launched (optional). The name may be prefixed by the domain name, for example "DOMAIN\USER".
'password'	A text, the password of the user.
Value:	A list of the following attributes:
	STATUS SCIL status code of the operation The following attributes are returned only if STATUS = 0 (OK)
	DA20_SERVERS A vector of server descriptors of the DA servers that support the version 2.0 of the standard.
	DA30_SERVERS A vector of server descriptors of the DA servers that support the version 3.0 of the standard.

The server descriptor is a list of the following three text attributes:

Table continues on next page

NAME	The name of the server
PROGID	The program ID of the server
CLSID	The class ID of the server

In the first form of the function call (i.e. the node is defined by a base system node number), the possible user account and its password are defined by the node object attribute OP (NODn:BOP). See the System Objects manual for the details.

## 10.25 RTU functions

The following functions apply only to S.P.I.D.E.R. RTUs.

### 10.25.1 RTU\_ADDR(key)

Returns a list with the address of the object in a certain record.

'key'	Text of three characters. The search key of the record.
Value:	A list with the following attributes:
TP	Integer, 0 ... 11. The type of the RTU200 object (see below).
BA	Integer. The block address in RTU200.

The TP attribute contains the four most significant bits and the BA attribute the 12 least significant bits of the object address (OA).

The TP attribute has the following meanings:

0	No object type
1	Object command
2	Regulation command
3	Digital set-point
4	Analog set-point
5	General persistent output
6	Analog value
7	Indication (single or double)
8	Pulse counter
9	Digital value
10	Indication event recording (indication with time stamp)
11	Analog event recording

Example:

```
@RTU = RTU_ADDR(RTU_KEY(X:POA1))  
;Now  
;RTU:VTP == 5  
;RTU:VBA == 201
```

### 10.25.2 RTU\_AINT(i)

Converts an integer to ASCII characters (according to the RP570 protocol).

'i'	An integer number.
Value:	A text of two characters.

**Example:**

```
RTU_HEXASC (RTU_AINT(342)) ; returns "0156"
```

### 10.25.3 RTU\_AREAL(r)

Converts a real number to four ASCII characters (float DS801).

'r'	A real number.
Value:	A text of four ASCII characters.

**Example:**

```
RTU_HEXASC (RTU_AREAL(5.5)) ; returns "40B00000"
```

### 10.25.4 RTU\_ATIME [(t [,msec])]

Converts time data (operating system time) to ASCII (RTU200 time).

't'	Time data. Default: the present time.
'msec'	The milliseconds of the time. Default = 0.
Value:	A text of 6 or 10 ASCII characters.

**Example:**

```
RTU_HEXASC (RTU_ATIME (OBJ:PRT)) ; returns "0EC5226ED990"
```

### 10.25.5 RTU\_BIN(h)

Converts HEX-ASCII numbers given as a text to binary numbers in text form.

'h'	A text or a text vector of HEX-ASCII numbers.
Value:	Text or text vector. The texts represent binary numbers.

**Example:**

```
RTU_BIN("414243") ; returns "ABC"
```

### 10.25.6 RTU\_HEXASC(b)

Converts binary numbers given as a text to hex-ascii numbers as a text.

'b'	A text or text vector representing 8 bit binary numbers.
Value:	A text of hex-ascii numbers.

**Example:**

```
RTU_HEXASC ("ABC") ; returns "414243"
```

## 10.25.7 RTU\_INT(a)

Converts two ASCII characters (2's complement RP570) to an integer.

'a'                                   A text of two ASCII characters.  
Value:                                 Integer.

Example:

```
RTU_INT(RTU_BIN("0156")) ; returns 342
```

## 10.25.8 RTU\_KEY(oa)

Returns the search key for a record in an RTU200 configuration file.

The process object corresponding to the record must be known.

'oa'                                   Integer. The object address, the OA attribute, of the process object stored in the record.  
Value:                                 A text of three characters. The search key for the record.

Example:

```
@UN = X:PUN1
#OPEN_FILE 1 0 "RTU'UN'.CFG" KL
#READ 1 RTU_KEY(X:POA1) RECORD
;%RECORD contains the configuration record associated with the object
X:P1.
```

## 10.25.9 RTU\_MSEC(atime)

Returns the milliseconds of the 6-byte RTU time string 'atime'.

'atime'                              Text. The RTU time.  
Value:                                 Integer. The number of milliseconds.

## 10.25.10 RTU\_OA(type, ba)

Returns the object address.

'type'                              Integer, 0 ... 11. The type of the object (see the TP attribute in the RTU\_ADDR description above).  
'ba'                                 Integer. The address (block address) of the object in RTU200, see the BA attribute in the RTU\_ADDR description above.  
Value:                                 Integer. The object address, the attribute OA of the process object.

Example:

```
RTU_OA(5,201) ; returns 20681
```

## 10.25.11 RTU\_REAL(a)

Converts 4 ASCII characters (float DS801) to a real number.

'a'                           A text of four ASCII characters.  
 Value:                        A real number.

**Example:**

```
RTU_REAL(RTU_BIN("40B00000")) ; returns 5.5
```

## 10.25.12 RTU\_TIME(a)

Converts ASCII (RTU200 time) to SYS600 time data.

'a'                           A text of 6 ASCII characters.  
 Value:                        Time data.

# 10.26 Printout functions

The functions described in this section are intended to be used only in printed pictures (format pictures).

## 10.26.1 PRINT\_TRANSPARENT(data [,log])

Sends printout to a printer.

The function sends a printout (data, printer commands and control codes) directly to printers defined as 'transparent' and writes data to the printer log file. The function enables full printer control and allows for freely formatted texts and graphics.

When called in a printed picture (printout activated by the #PRINT or #LIST command or by an event in the process database), the function spools the output (printer commands, control codes and data) to the printer (see [Figure 18](#)).

The function should be placed in the start program of all pictures used for printout on 'transparent' printers. If the printed picture contains picture functions, the start programs are executed in the following order: the start program of the main picture, the start program of the base function, the start programs of the picture functions in the installation order. When used in other environments than printed pictures (pictures shown on screen, command procedures), the function causes an error message (SCIL\_UNKNOWN\_FUNCTION).

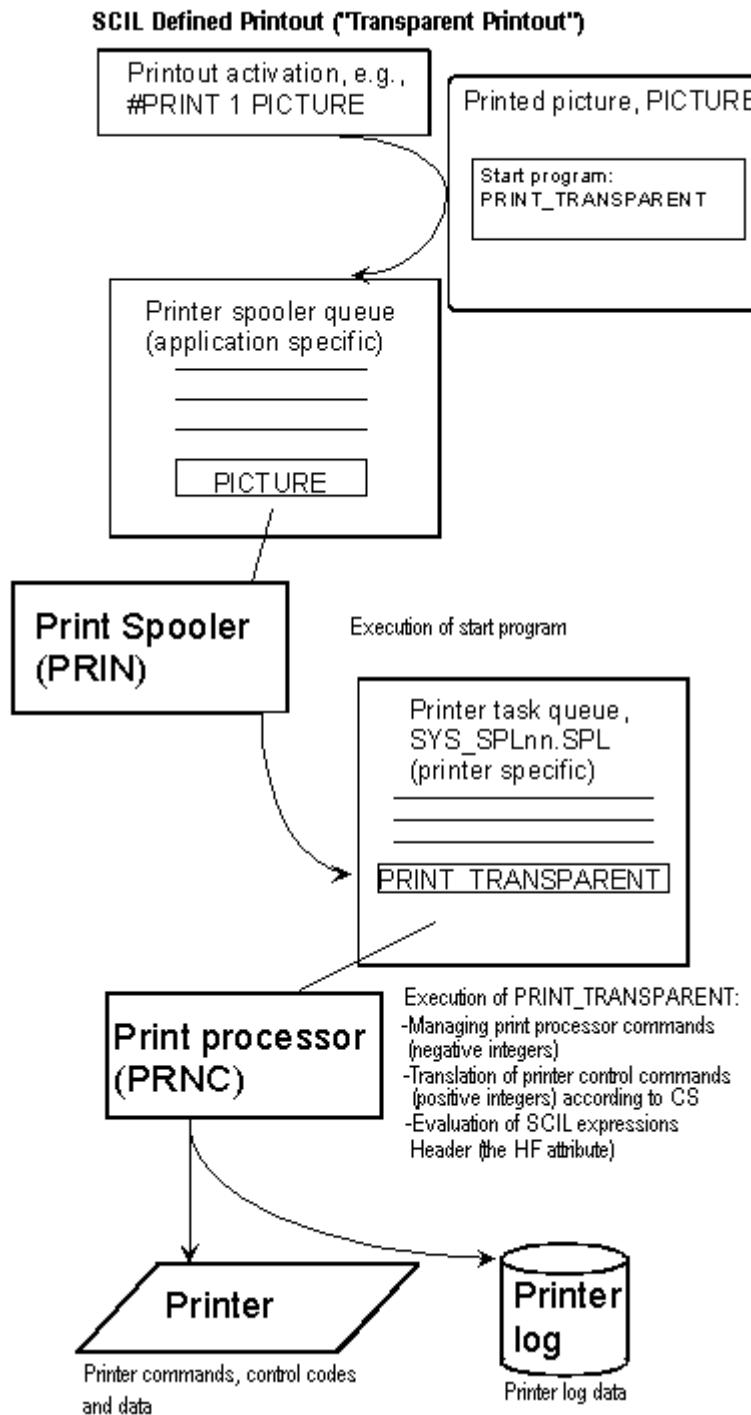


Figure 18: SCIL defined or "transparent" printout

The target printer(s) of the printout are defined by the initiating #PRINT or #LIST command or the LD attribute of the process object. If there is no transparent printer among the target printers, the function has no effect. Printers defined as 'transparent' can only handle printout defined by this function.

Unlike the semi-graphic picture based printout, the printout produced with the PRINT\_TRANSPARENT function does not automatically include any SYS600 picture elements, only the texts, graphics and formatting specified by the function. However, complete pictures and picture elements may be included as printout files.

The first argument of the function, 'data', specifies the printout - i.e. the printed data, formatting and print processing. The second (optional) argument, 'log', specifies the output to the log file. The function returns the status of the printer spool operation.

'data' Vector value, the printout vector, or integer 0 = no printout.  
The printout vector is an ordinary SCIL vector containing text, integer and vector type elements. It may contain the following elements:

- The data to be sent to the printer. This is specified by texts and text vectors given as constants or SCIL expressions. The texts may be encoded in any symbol set supported by the printer. The SCIL expressions must be marked by the printout processing command -5 as the preceding element (see below). They are evaluated at the moment of physical printing, and the result is sent to the printer. Expressions can be used, e.g. to print the actual printout time of a report. The text elements may also comprise printer control sequences for producing graphics. However, this is generally not recommended, because they would make the print vector printer interface dependent and the print processor would not be able to keep track of output pages. Instead, printer control sequences can be defined in the CS attribute and included in the print vector as printer control commands. Files (e.g. window dump files transferred to printout files) may be included using the READ\_TEXT function.
- Printer control commands. The printer control commands are logical commands that control the printer. They are represented as positive integers defined by the printer specific CS attribute of the printer object (see the System Objects manual). There is a tool for defining the printer control commands.
- Print processor commands. The print processor commands control the processing of the print vector itself. They are given as negative integers. The following print processor commands are available:

-1	Auto-NL on	Automatic new line feature on (default)
-2	Auto-NL off	Automatic new line feature off
-3	Increment LN	Increment line number attribute LN
-4	Increment PN	Increment page number attribute PN
-5	Formula follows	The next element in the vector will be interpreted as a SCIL expression instead of plain text

The automatic new line feature (on by default) means that a new line command (printer control command 1, see above) is automatically appended to each text element in the print vector.

The increment LN and increment PN commands may be used to inform the print processor that a new line or new page is started using printer control commands that are not predefined (see the CS attribute).

'log' Vector (or integer 0). The contents of the vector is in full transferred to the log file (provided that the printer has been defined with printer log output in the base system configuration). SCIL expressions are evaluated before the transfer. 0 = no log file output.

Value: Integer. The status code generated by the print spool operation. 0 = OK or no 'transparent' printer among the target printers.

The target printers can be examined by means of the PRINTER\_SET function (see below). By reading the SCIL\_HOST function, the SCIL program can determine whether the picture is being printed or displayed on the screen.

Example:

```
#LOCAL HOST, OX, CX, OV, S
HOST = SCIL_HOST
#IF HOST.NAME == "PRIN" #THEN #BLOCK
```

```
OX = 'LN':POX'IX'
CX = 'LN':PCX'IX'
OV = 'LN':POV'IX'
S = PRINT_TRANSPARENT((-2,TIMES, "OBJECT TEXT:",OX,-
                        "COMMENT TEXT:",CX, " OBJECT VALUE:",-1,DEC(OV)))
#BLOCK_END
```

This program block first checks whether the picture is being printed or shown on the screen. If it is printed, the PRINT\_TRANSPARENT function prints a row containing the present time and the values of attributes OX, CX and OV of a process object.

## 10.26.2 PRINTER\_SET

Returns the target printer numbers.

Value: A vector of integer elements.

The function returns the logical target printer numbers (mapped through APL:BPR to find the physical printer numbers) of the printed picture. By means of this function, the SCIL program may examine the properties of target printers.

Called somewhere else than in a printed picture, the function has no meaning.

## 10.27 User session functions

The following functions are used manage a user session and read its attributes. A session begins by login and ends either by logout or termination of the client program. Most of the functions can be used only during an active session.

### 10.27.1 SET\_EVENT\_LIST\_USER\_NAME(name)

Sets the name that is displayed in User Name column of the event list.

'name'	Text	The name that is used as the User Name in the Event List for events generated by the calling SCIL program. In the Event List the name appears prefixed by * for distinction between these artificial and proper user names.
Value:	Integer	SCIL status code:
	0	OK_STATUS
	8033	USER_ALREADY_LOGGED_IN This function may not be used by a logged-in user.

This function is used by SCIL programs that run without a login (for example event and time channel command procedures).

OPC Clients that run without a login appear as "OPCGuest" in the event list. The user name OPCGuest is a reserved name that may not be used as a real user name.

The event list user name of external SCIL-API programs is of the format "SCIL-API: username", where the 'username' is the Windows user name of the launcher of the SCIL-API program.

## 10.27.2 USM\_ADDRESS

Tells the address of the workplace computer where the user logged in.

Value:	Text	Address of the session, either computer name or IP address. Empty string if not logged in.
--------	------	--

## 10.27.3 USM\_AOR\_DATA

Reads the AoR(Area of Responsibility) related data of the user session.

Value:	Vector	Vector of lists with following attributes (empty vector if the user is not an AoR operator):
	AOR	Name of the area
	ROLE	Name of the area role of the user
	USERS	Vector of lists describing the other users of the area, see below
		The following two attributes are given only when Exclusive Access Rights (EAR) are in use:
	EAR_ROLE	Integer defining the EAR role of the user 0 = Viewer 1 = Secondary operator 2 = Primary operator 3 = Master operator
	EAR_OWNER	Name of the EAR owner user
		The list describing other users contains the following attributes:
	USER_NAME	Name of user
	EAR_ROLE	EAR role of user (only when EAR is in use)
	ROLE	Area role of user
	LOGGED_IN	Boolean

## 10.27.4 USM\_AUTHORIZATION\_LEVEL(group)

Reads the authorization level of the user in a given authorization group.

'group'	Text	The name of the authorization group
Value:	Integer	The user's authorization level in the group 0 ... 5, or -1 if not logged in

## 10.27.5 USM\_AUTHORIZATION\_LEVEL\_FOR\_OBJECT(group, object\_name, object\_index)

Reads the authorization level of the user for controlling a process object.

'group'	Text	The name of the authorization group used for process object control
'object_name'	Text	The name of the process object to be controlled
'object_index'	Integer	The index of the process object to be controlled
Value:	Integer	The user's authorization level for the process object 0 ... 5, or -1 if no access

This function is used to determine user's authorization level for controlling a process object when Area of Responsibility (AoR) concept is in use.

The area is determined by the Object Identifier (OI) attribute of the process object. If the user has a role in the area, authorization level of that role is returned, otherwise -1 (no access) is returned. If the process object is contained in more than one area (overlapping areas), the highest authorization level in the user's area roles is returned.

If Exclusive Access Rights (EAR) concept is in use, only the current EAR owner is given the 'real' authorization level. Other AoR operators are given authorization level 0.

If AoR concept is not used or the given process object is not contained in any area, the user's authorization level in the group is returned according to the session role.

## 10.27.6 USM\_AUTHORIZATIONS

Reads the authorizations of current user.

Value:	List vector	Vector of lists with following attributes (empty vector if not logged in):
		GROUP              Authorization group name
		LEVEL              Authorization level for the group

## 10.27.7 USM\_CHANGE\_PASSWORD(*old\_password*, *new\_password*)

Changes the password of the user.

'old_password'	Text	Old password
'new_password'	Text	New password
Value:	Integer	SCIL status code:
	0	OK_STATUS
	8035	USER_INVALID_USER_OR_PASSWORD The old password is wrong.
	8017	USER_PASSWORD_POLICY_VIOLATION The new password is not valid.
	8034	USER_NOT_LOGGED_IN
	8036	USER_SESSION_NOT_VALID The session has timed out.

## 10.27.8 USM\_IS\_NEW\_APPLICATION

Tells whether the current application is a new one.

Value: Boolean TRUE or FALSE

The word 'new' here means that the next user that logs in to the application becomes an Administrator user of the application. This function may be called before login.

## 10.27.9 USM\_LOGIN(name, password)

Logs in a user starting a new session or silently joins an existing session as a sub-session.

'name'	Text	Name of the user. If this is a silent login to join an existing session, the id of the session is given as 'name'.
'password'	Text	Password of the user. If this is a silent login, password is empty ("").
Value:	Integer	SCIL status code:
	0	OK_STATUS
	8033	USER_ALREADY_LOGGED_IN The user is already logged in.
	8049	USER_LOGIN_FAIL_WRONG_CR Wrong credentials (user name or password)
	8051	USER_LOGIN_FAIL_EXPIRED The password has expired.
	8052	USER_LOGIN_FAIL_LOCKED The user account has been locked because of too many retries.
	8053	USER_LOGIN_FAIL_DISABLED The user account has been disabled.
	8055	USER_LOGIN_FAIL_NO_CAM No connection to Centralized Account Management (CAM)

Obviously, this function is called before login.

## 10.27.10 USM\_LOGOUT

Logs out the user.

Value:	Integer	SCIL status code:
	0	OK_STATUS
	8034	USER_NOT_LOGGED_IN
	8036	USER_SESSION_NOT_VALID Session has timed out.

After logout, the SCIL program runs in read-only mode.

## 10.27.11 USM\_PASSWORD\_CHANGE

Tells the status of the password.

Value:	Text	One of the following keywords:
	""	No need to change password.
	"FIRST_LOGIN"	New user account. Administrator has forced a password change.
	"RESET"	Administrator has given a new password and forced a password change.
	"EXPIRED"	Password has expired and must be changed.
	"EMERGENCY_LOGIN"	This is an emergency login. A proper permanent password must be given.

## 10.27.12 USM\_PASSWORD\_POLICY

Reads the password policy of the application.

Value:	List	The following attributes (empty list if not logged in):
	IU	In use, one of the following values:
	0	Password policy is not applied
	1	Password policy is applied
	-1	Not known. The contents of the policy and its use is not known. This is the case, when CAM is used for authentication.
		The following attributes are listed only when IU is 1.
	PL	Minimum password length
	UC	Minimum number of upper case characters
	LC	Minimum number of lower case characters
	NC	Minimum number of numeric characters (digits)
	SC	Minimum number of special characters (other than UC, LC or NC characters)

## 10.27.13 USM\_SELECT\_ROLE(role)

Selects the role for the starting session.

'role'	Text	The name of the role to be selected
Value:	Integer	SCIL status code:
	0	OK_STATUS
	8034	USER_NOT_LOGGED_IN
	8036	USER_SESSION_NOT_VALID The session has timed out.
	8002	USER_ROLE_CAN_BE_SELECTED_ONLY_AT_LOGIN The role cannot be selected twice.

## 10.27.14 USM\_SESSION\_ATTRIBUTES

Reads the Monitor Pro session attributes.

Value:	List	The following attributes (empty list if not logged in):
		MONITOR_PRO_SESSION_TIMEOUT
		MONITOR_PRO_INACTIVITY_TIMEOUT

## 10.27.15 USM\_SESSION\_ID

Tells the id of current session.

Value:	Text	Id of current session. Empty string if not logged in.
--------	------	---

## 10.27.16 USM\_SESSIONS

Lists the active sessions in the system.

Value:	A list vector of active sessions. Each element describes a session by the following attributes:		
	APL	Integer	Application number
	NR	Integer	Sequence number of the session
	USER	Text	Name of the user
	ROLE	Text	Name of the user's role in the session
	ADDRESS	Text	Name or IP address of the workplace computer
	LOGIN_TIME	Time	Local time of the login
	BREAK_TIME	Time	Local time of the connection break between kernel and workplace
	SUBSESSIONS	Integer	Number of sub-sessions in the session
	READ_ONLY	Boolean	TRUE, if the session runs in read-only mode

This function requires super-user access rights (authorization level of group GENERAL must be 5).

## 10.27.17 **USM\_USER\_LANGUAGE**

Tells the user's preferred language.

Value:	Text	Two-letter ISO 639-1 abbreviation of the preferred language of the user. Empty string if not logged in.
--------	------	---

## 10.27.18 **USM\_USER\_NAME**

Tells the name of current user.

Value:	Text	Name of the user. Empty string if not logged in.
--------	------	--

## 10.27.19 **USM\_USER\_ROLE**

Tells the name of the role that the user has selected at login.

Value:	Text	The role of the in this session. Empty string if not logged in.
--------	------	---

## 10.27.20 **USM\_USER\_ROLES**

Lists the roles that the user may select from after a successful login.

Value:	Text vector	Names of roles. Empty vector if not logged in.
--------	-------------	--

## 10.27.21 **USM\_USER\_SESSION\_DATA**

Lists the attributes of the current user session.

Value:	A list of current user session attributes:		
	LANGUAGE	Text	Two-letter ISO 639-1 abbreviation of the preferred language of the user
	SESSION_ATTRIBUTES	List	Integer attributes MONITOR_PRO_SESSION_TIMEOUT and MONITOR_PRO_INACTIVITY_TIMEOUT
	SESSION_ID	Text	Id of current session
	USER_NAME	Text	Name of the user
	USER_ROLES	Text vector	Name of roles that the user can have
	LOGIN_TIME_UTC	Time	UTC time of login

## 10.28 Miscellaneous functions

### 10.28.1 **ADD\_INTERLOCKED(object, index, amount)**

Modifies the UV or SV attribute of a SYS or an APL object.

ADD\_INTERLOCKED supports synchronization of SCIL programs executing in parallel.

'object'	Text keyword value, either "SYS" or "APL". Specifies the base system object. Note that application number may not be given, the current application is always assumed.
'index'	<p>Integer or vector value.</p> <p>If an integer, specifies the index of the UV attribute to be modified.</p> <p>If a vector, specifies both the attribute and the index: The first element is a text value, either "UV" or "SV". The second element is an integer value specifying the index.</p>
'amount'	Integer value to be added into the element of the UV or SV attribute.
Value:	Integer value, the result of the addition.

Example:

```
#LOCAL DUMMY
#LOOP ADD_INTERLOCKED("SYS",("SV",77),1) <> 1 ;Try to reserve the resource
        DUMMY = ADD_INTERLOCKED("SYS",("SV",77),-1) ;Didn't get it
        #PAUSE 0.5 ;Wait a while
#LOOP_END ;and try again
;The resource is now reserved for exclusive use
DUMMY = ADD_INTERLOCKED("SYS",("SV",77),-1) ;Release the resource
```

A system wide (inter-application) binary semaphore may be implemented by this SCIL code. SYS:BSV77 is used as the semaphore. It is supposed to be set to zero in SYS\_BASCON.COM. The value returned by 2nd and 3rd call of the function is not used anywhere. For clarity, the code to handle a deadlock situation is omitted. In a real application, too long waits must be avoided.

## 10.28.2 AUDIO\_ALARM(alarm\_class, on\_or\_off)

Sets and resets the specified audio alarm(s).

'alarm_class'	Integer value 1 ... 8 or text keyword "ALL"
'on_or_off'	Text keyword "ON" or "OFF"
Value:	Integer, the status code of the operation, (0=OK_STATUS).

The value "ALL" for the argument 'alarm\_class' sets or resets the audio alarm for all alarm classes 1 to 7.

The value 8 is used to force the watchdog flip-flop bit to 0 or 1 at system start-up or shutdown.

## 10.28.3 SCALE(v, scale\_object [,direction])

Scales a value using a scale object.

'v'	Real value, the value to be scaled.
'scale_object'	Text value, the name of the scale object to do the scaling.
'direction'	Text value, either "INPUT" or "OUTPUT". Defaults to "INPUT".
Value:	Real value, the scaled value.

The direction "INPUT" corresponds to the scaling of analog input process objects, direction "OUTPUT" corresponds to the scaling of analog output process objects.

Example:

```
@r_invalue=SCALE(100,"TEST_SCALE","INPUT")
@r_outvalue =SCALE(0.1,"TEST_SCALE","OUTPUT")
```

## 10.28.4 UNLOCK\_PICTURE(picture)

Unlocks a locked picture.

'picture'	Text value, the name of the picture in SCIL ([path/]name) or operating system file name format.
Value:	Integer value, a SCIL status code.
0	OK_STATUS
4024	PICF_PICTURE_IS_NOT_LOCKED
	Other status codes may also be returned if the argument is invalid.



This function should be used only after a monitor process has crashed or silently disappeared while the picture has been displayed or edited in the monitor, leaving the picture locked. In the locked state a picture cannot be edited or in some cases not even displayed, error 4005 (PICF\_SHARE\_ERROR) is raised instead.

# Section 11      Graphics primitives

This section describes the use of the graphics primitives, which constitute the base of all full graphics elements in SYS600 pictures, partly also in dialogs. The first section introduces some concepts related to the graphics primitives.

## 11.1      Introduction

SYS600 full graphics in pictures, partly also in dialogs, are realized using SCIL graphics commands. There are SCIL commands for drawing various types of graphical elements, such as points, lines, polylines and polygons, arcs, circles, ellipses, boxes (rectangles), and texts. The SCIL graphics commands specify the geometry of the graphical elements, while the location and size, as well as other features, are given as arguments.

The graphics commands can be included in any SYS600 picture program (see [Section 4.1](#)) and in the methods of the dialogs and dialog items. The BACKGROUND program of a picture contains the graphics commands generated by the picture editor. As a rule, this program should not be edited manually. Context dependent graphics in the background can be written in the DRAW program. Like picture commands and Visual SCIL commands ([Section 9](#)), the graphics commands are not allowed in command procedures, unless they are executed by #DO commands or DO functions situated in user interface objects. The graphics commands are described in [Section 11.2](#).

### 11.1.1      Graphics contexts

The features of the graphical elements (color, type of line, line width, font, etc.) are defined by graphics contexts, which are identified by integer numbers. A graphics context is a series of properties, called "components", such as the ones mentioned. Several commands can use the same graphics context, though all properties are not used by all graphics commands. The graphics contexts are described in [Section 11.3](#).

### 11.1.2      Graphics canvas

The graphical elements are displayed in the user interface object (picture, picture function, dialog item) chosen as graphics "canvas". By default, the canvas is the picture or Visual SCIL object where the graphical commands are situated.

Any picture and picture function can be chosen as graphics canvas. However, regarding the dialogs, only certain types of dialog items can contain graphics elements. These are the dialog items that allow the display of images (have the image attribute).

The location of a graphical element is given by x,y coordinates related to the home position of the canvas. The upper left corner of the picture or picture function where the element will be displayed. As the resolution of screens varies, the graphics commands use a screen independent coordinate system. The coordinate system can contain SCIL coordinates or VS coordinates. The SCIL coordinate system is fixed by a scaling factor. By means of an input command, the coordinates can be read from the mouse position. Refer to [Section 11.4](#). to learn about the SCIL and VS coordinates, and the mouse input commands.

### 11.1.3 Miscellaneous

The context definitions, canvas selection and scaling can be temporarily stored and restored within the same picture, see [Section 11.5](#).

The display of the graphics on screen can be controlled by some display handling commands ([Section 11.5.2](#)).

## 11.2 Full graphics SCIL commands

### 11.2.1 Drawing graphical elements

The commands below draw graphical elements on screen, in the user interface object selected for canvas (default = the same object as where the commands are situated). Coordinate system is different in pictures and in VS objects. The command .COORDINATE\_SYSTEM is described in [Section 11.4](#).

The color and other features of the elements are fixed by the graphics contexts. The graphics contexts are described in [Section 11.3](#).

In most cases, several elements can be drawn with one command by giving vectors for coordinates and sizes. If one coordinate or size is a scalar while the other one is a vector, the scalar argument is handled as a vector with all elements equal, for example .POINT 0, (100,200) is equivalent to .POINT (0,0), (100,200). If the coordinates fall outside the canvas (that is they are negative or too large), they will still affect the shape of the graphical element. The part of the graphical element that falls outside the canvas will not be displayed.

In the command syntax descriptions below, all arguments given in lower case letters can be SCIL expressions of the data type required for the respective argument.

#### 11.2.1.1 .ARC [[scope,]n :] x,y, r, a1,a2 [,FILL]

Draws one or several arcs as circle segment arcs.

'scope'	The scope of the context (see <a href="#">Section 11.3</a> ).
'n'	Graphics context number, 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, DASH_OFFSET, DASH_LIST, ARC_MODE. See <a href="#">Section 11.3</a> .
'x,y'	The coordinates of the center of the circle. Real, integer or vector of real and integer elements.
'r'	The radius of the circle given in coordinate units, see <a href="#">Section 11.4</a> . Real, integer or vector of real and integer elements.
'a1'	The angle from where the arc starts, given in degrees increasing counter-clockwise from horizontal 0 value. Real, integer or vector of real and integer elements. See <a href="#">Figure 19</a> .
'a2'	The angle of the arc given in degrees. If a2 is positive, the arc is drawn counter-clockwise. Otherwise, it is drawn clockwise. Real, Integer or vector of real and integer elements. See <a href="#">Figure 19</a> .

If the FILL option is given, the arc is filled according to the ARC\_MODE component of the graphics context.

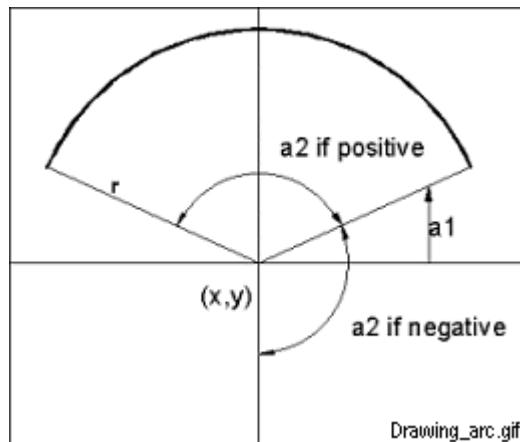


Figure 19: Drawing an arc with the .ARC command

#### 11.2.1.2 .BOX [[scope,]n :] x,y, width, height [,FILL]

Draws one or several boxes (rectangles).

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, JOIN_STYLE, DASH_OFFSET, DASH_LIST. See <a href="#">Section 11.3</a> .
'x,y'	Integer or real, or vector of integer or real data. The coordinates of the upper left corner of the rectangle.
'width'	Integer or real or vector of integer or real data. The width of the rectangle given in coordinate units, see <a href="#">Section 11.4</a> .
'height'	Integer or real or vector of integer or real data. The height of the rectangle given in coordinate units, see <a href="#">Section 11.4</a> .

If the FILL option is given, the box is filled (in the FOREGROUND color).

#### 11.2.1.3 .CIRCLE [[scope,]n :] x,y, r [,FILL]

Draws one or several circles.

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, DASH_OFFSET, DASH_LIST. See <a href="#">Section 11.3</a> .
'x,y'	Integer or real data, or integer or real vector. The coordinates for the center of the circle.
'r'	Integer or real data, or integer or real vector. The radius given in coordinate units, see <a href="#">Section 11.4</a> .

If the FILL option is given, the circle is filled.

#### 11.2.1.4 .ELLIPSE [[scope,]n :] x,y, a,b [,FILL]

Draws one or more ellipses.

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, DASH_OFFSET, DASH_LIST
'x,y'	Integer or real data, or integer or real vectors. The coordinates of the center of the ellipse.
'a,b'	Integer or real data, or integer or real vectors. The axes of the ellipse given in coordinate units, see <a href="#">Section 11.4</a> .

If the FILL option is given, the ellipse is filled.

#### 11.2.1.5 .IMAGE x, y, w, h, filename, tag\_1[, tag\_2[, tag\_3[, tag\_4]]]

.IMAGE command draws a VS\_IMAGE object into the current window or VS object.

'x'	Real, x coordinate
'y'	Real, y coordinate
'w'	Real, width, 0 ... 32 767
'h'	Real, height, 0 ... 32 767
'filename'	Text, vso file name
'tag_1'	Text, tag name within the vso file, image to be used when 8 x 10 font is used.
'tag_2'	Text, tag name within the vso file, image to be used when 12 x 15 font is used.
'tag_3'	Text, tag name within the vso file, image to be used when 16 x 20 font is used.
'tag_4'	Text, tag name within the vso file, image to be used when 20 x 25 font is used.

The current coordinate system is SCIL, the coordinates x and y define the position of the upper left corner of the image, otherwise the lower left corner.

'w' and 'h' specify the size of the rectangle the image is drawn into, the image is scaled to fill the rectangle.

If 'w' is 0, no horizontal scaling is done. If 'h' is 0, no vertical scaling is done.

'filename' specifies the VSO file where the image is stored.

At least one image must and up to 4 images may be specified in the command. The one used in drawing depends on the size of the current MicroSCADA font. If no image is specified for the current font size, the image for the next smaller font is used. If there is no image for smaller fonts, the image for the next larger font is used. When an image for a wrong font size is used, it is scaled in proportion to font sizes (if 'w' and/or 'h' is 0).

#### 11.2.1.6 .LINE [[scope,]n : ] x1,y1, x2,y2

Draws one or more lines from (x1,y1) to (x2,y2).

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope. Default = 0 Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, DASH_OFFSET, DASH_LIST. See <a href="#">Section 11.3</a> .
'x1,y1'	Integer or real, or vector of integer or real data. The coordinates of the start point. See <a href="#">Section 11.4</a> .
'x2,y2'	Integer or real, or vector of integer or real data. The coordinates of the end point. See <a href="#">Section 11.4</a> .

### 11.2.1.7 .POINT [[scope,]n :] x,y, [RELATIVE]

Draws one or several points of one pixel's size at the coordinates (x,y). If any of x or y or both are vectors, and the option RELATIVE is given, the first point is taken as canvas relative, the next one as relative to the previous one, etc.

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope. Default: 0. Used components: FUNCTION, FOREGROUND. See <a href="#">Section 11.3</a> .
'x,y'	Integer or real, or vector of integer or real data. coordinates for the point, see <a href="#">Section 11.4</a> .

### 11.2.1.8 .POLYLINE [[scope,]n :] x,y [,RELATIVE] [,FILL]

Draws a polyline or a polygon.

'scope'	is the scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, LINE_WIDTH, LINE_STYLE, CAP_STYLE, JOIN_STYLE, DASH_OFFSET, DASH_LIST.
'x'	real or integer, or vector with real and integer elements, the x-coordinates in SCIL or VS coordinate units.
'y'	real or integer, or vector of real and integers, the y-coordinates in SCIL or VS coordinate units.

The corresponding elements in the x and y vectors are taken as x,y-coordinates (see [Figure 20](#)). The command draws a line between each of the coordinates.

If the RELATIVE option is included in the argument list, the first coordinate pair is placed relative to the upper left corner of the canvas and each of the following coordinates are drawn relative to the previous one.

If the first and last point coincide, the polyline will form a closed figure (a polygon), where the end points are joined according to the JOIN\_STYLE component of the graphical context.

If the FILL option is included in the argument list, the polygon outlined will be filled (with the FOREGROUND color). In this case, the polyline is always closed.

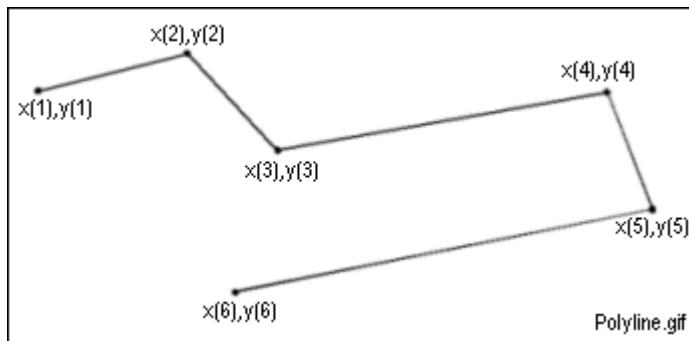


Figure 20: Drawing a polyline with .POLYLINE

#### 11.2.1.9 .TEXT [[scope,]n :] x,y, text [,FILL] [,align]

This command draws horizontal text starting at (x,y). The size and font of the texts is determined by the graphics context.

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer expression 0 ... 20 or 0 ... 50 depending on the scope. Default = 0. Used components: FUNCTION, FOREGROUND, BACKGROUND, FONT. See <a href="#">Section 11.3</a> .
'x,y'	Integer, real or vector of integer or real. The coordinates of the start point of the text. The starting-point (x,y) defines the location of the origin pixel of the first character (if left alignment). The location of the origin pixel is font dependent. For the MicroSCADA semi-graphic font, it is the upper left corner of a character.
'text'	A SCIL expression of type text or a text vector.
'align'	The alignment of the text: LEFT, RIGHT or CENTER. The default is LEFT, if the font is a left-to-right font, and RIGHT if the font is a right-to-left font (e.g. Hebrew).

When multiple lines are drawn, i.e. 'text' is a text vector, the x,y coordinates can be vectors with the coordinates for each new line or they can be scalars. In the latter case, the text lines are displayed under one another starting from x,y with a line space adjusted to the font characteristics.

If the FILL option is given, the background of the text is filled with the BACKGROUND color of the graphics context, otherwise the background is transparent.

## 11.3 Graphics contexts

### 11.3.1 General

A graphics context is a collection of components ([Section 11.3.3](#)) each of which define a graphical property, for instance, background color, foreground color, font. A picture contains a number of graphical contexts identified by a number. The context numbers are used as arguments in the graphical commands. The maximum number of graphics contexts that can be used in one picture is limited to 2 000.

The defined graphics contexts (together with the canvas selection and the scaling) can be temporarily stored and restored, see [Section 11.5](#).

### 11.3.1.1 Scope of graphics contexts

The scope of a graphics context is the set of objects where the context applies. A graphics context can be defined for a single object (picture, dialog object), for a group of objects, or for all objects used in the monitor. There are five different scopes, each of which can contain a number of graphics context definitions. In the context definition commands, the scope is given by a text, a single letter or a word as follows:

"C" or "CURRENT"	The scope is the object (picture or Visual SCIL object) containing the .GC command. Max. number of contexts: 20.
"P" or "PARENT"	The scope is the parent object of the object in which the context is defined with .GC. Max. number of contexts: 20.
"R" or "ROOT"	The scope is the root object (main dialog, picture container or main picture). Max. number of contexts: 20.
"M" or "MONITOR"	The scope is all object used in the monitor. Max. number of contexts: 50. This scope is reserved for LIB 500 pictures.
"U" or "USER"	The scope = all objects shown in the monitor. Max. number of contexts: 50. This scope is for use in the application pictures.

The number of the context together with its scope identify the context and distinguishes it from other contexts with different context number or scope.

When a part picture is shown, it inherits the contexts of its parent picture. However, changes in the contexts of the parent picture do not affect the contexts of the part pictures which are already displayed on screen. Likewise, the contexts are inherited downwards in the hierarchy of a dialog system.

### 11.3.1.2 Default settings

As long as no component definitions have been done for a context in a certain scope, all the components have the default values mentioned in the component descriptions [Section 11.3.3](#). The contexts (except number 99) can be modified any time with the commands described in [Section 11.3.2](#).

Context number 99 contains the default settings of the components. This contexts cannot be changed. It contains the following components:

FUNCTION	"COPY"
FOREGROUND	"WHITE"
BACKGROUND	"BLACK"
LINE_WIDTH	0
LINE_STYLE	"SOLID"
CAP_STYLE	"BUTT"
JOIN_STYLE	"MITER"
FONT	"" (semi-graphic)
ARC_MODE	"PIESLICE"
NAME	""

GC number 99 may only be used as a source in GC copy. For self-documentation, a predefined constant name DEFAULT\_GC may be used instead of number 99 (DEFAULT\_GC == 99).

Context number 0 is the default context in those cases where no context number is given in the graphics commands.

## 11.3.2 Defining graphics contexts

The following three commands are used for the modification of graphics contexts. The graphics contexts can be read with a SCIL function described in [Section 11.3.5](#).

### 11.3.2.1 .GC [[scope,]n [=scope,]m]:[[component = value]...[,component = value]]

Modifies the graphics context number 'n'.

'scope'	A text which specifies the scope of the context, i.e., the objects (pictures, and Visual SCIL objects) where the context is valid. 'scope' can take the values "C" or "CURRENT", "P" or "PARENT", "R" or "ROOT", "M" or "MONITOR", "U" or "USER". See <a href="#">Section 11.3.1</a> . Default value = "C".
'n'	The number of the graphics context, integer 0 ... 20 or 0 .. 50. This number is used in the graphical commands, see <a href="#">Section 11.2</a> . Default: 0.
'm'	The number of another graphics context which is copied to the current one before modification. Integer, 0 ... 20 or 0 ... 50.
'component'	The name of a component to be modified. The component names can be given in the complete or abbreviated form, see <a href="#">Section 11.3.3</a> .
'value'	A SCIL expression, the value assigned to the component. The allowed data types are mentioned in the component descriptions in <a href="#">Section 11.3.3</a> .

The modification concerns all graphical elements subsequently drawn with the context in question. It does not affect the graphical elements drawn previously with the same context number. Those components which are not included in the component list are not affected by the command. The graphics contexts are inherited to the pictures shown in windows and to the picture functions.

The ROOT, MONITOR and USER scopes represent separate sets of contexts, while the CURRENT and PARENT scopes can mean the same or separate sets of contexts depending on the situation.

Example:

("C",5), ("R",5), ("M",5) and ("U",5) are all different graphics contexts. The contexts ("C",5) and ("P",5) may be the same or different contexts. When a part picture is shown, its context is the same as the one of the parent (i.e. CURRENT = PARENT). If the parent's context is changed later, they become two different contexts and the parent's context is referred to as ("P",5).

The PARENT scope could, e.g., be used in a general purpose tool designed to change the appearance of any picture it is used in. The MONITOR and USER scopes are suitable for defining monitor specific graphics contexts in the start program of the APL\_INIT picture, which is shown each time a monitor is mapped for an application (either at system start-up or later).

Example:

```
.GC 0 = DEFAULT_GC : FOREGROUND = "RED"
```

## 11.3.3 Components of graphics contexts

This sub-section describes the components included in the graphics contexts. The components are listed in alphabetical order.

The components are assigned values by means of the commands described in [Section 11.3.2](#). The values are given as SCIL expressions of the allowed data types. From the start, the components have the default values given below. In the context defining commands, the

component names can be given in complete form or in the abbreviated form found to the left in the description below.

### 11.3.3.1 AM ARC\_MODE

This component specifies how filled arcs are drawn, either as sectors or segments.

Value:	"CHORD"	The end points of the arc are connected and the resulting segment is filled.
	"PIESLICE"	The end points of the arc are connected in the center of the circle and the resulting sector is filled.
Default:	"PIESLICE"	

### 11.3.3.2 BG BACKGROUND

Specifies the "background color" of the full graphic lines and text. This BACKGROUND color is used in the following cases:

- In the gaps of double dashed lines
- As the background of "filled" texts

Values:	The color can be given in the following four ways (see <a href="#">Section 11.3.4</a> ):
	With a color name given as a text, e.g., "LIGHT BLUE"
	With an RGB number given as a vector of three elements, e.g., (10000,20000,30000)
	With a color number given as an integer expression, e.g. 5
	With scope and color number given as a vector of two elements: (scope,color_number), e.g., ("M",5).
Default value:	"BLACK"

### 11.3.3.3 CS CAP\_STYLE

Specifies the endpoints of the line, see [Figure 22](#).

Values:	"BUTT"	The line is cut off in a 90° angle to the line direction at the end point.
	"NOTLAST"	The same as "BUTT", but the line is cut off one pixel before the end point.
	"PROJECTING"	The line is cut off in the same way as "BUTT" half its width past the endpoint.
	"ROUND"	The line end is rounded in a half circle with r = half its width past the endpoint.
Default value:	"BUTT"	



"PROJECTING" and "ROUND" are meaningful only for lines with a width larger than 1 pixel.

### 11.3.3.4 DL DASH\_LIST

Specifies the length of dashes and gaps when drawing dashed lines.

Value:

Vector of even length. The odd indexes define the lengths of dashes and the even indexes the lengths of the gaps. The lengths are given in coordinate units, see [Section 11.4](#).

Default:

The dashes as well as the gaps are four pixels long.

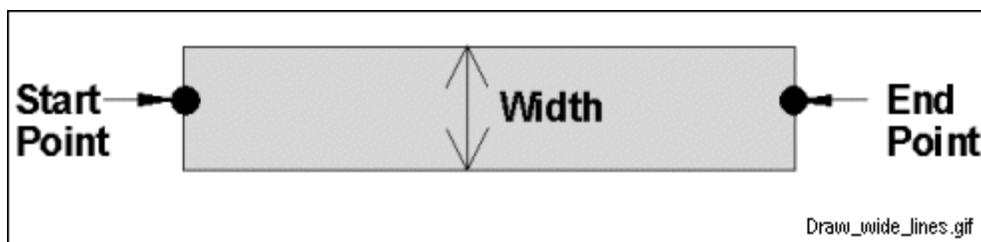


Figure 21: Wide lines are drawn centered in relation to the start and end points

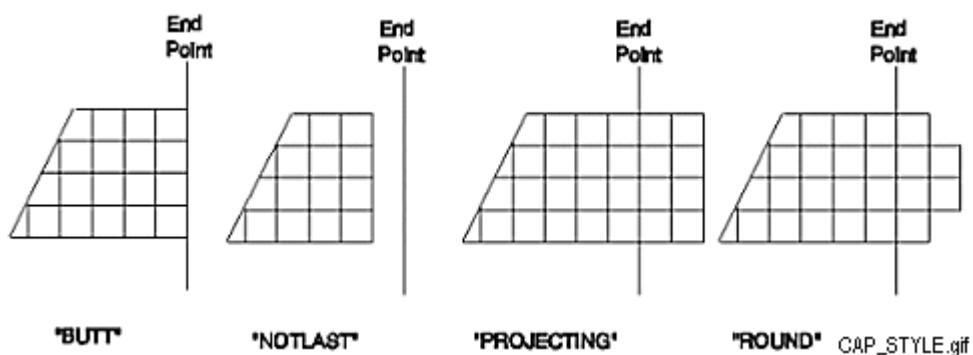


Figure 22: The CAP\_STYLE component

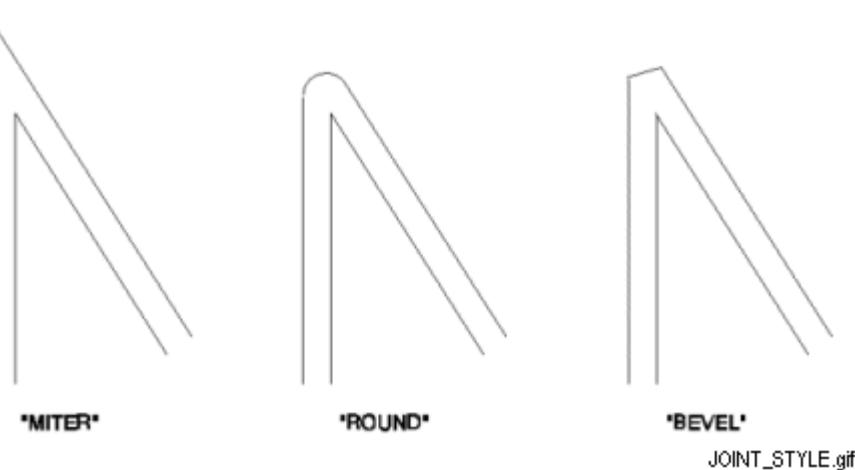


Figure 23: The JOIN\_STYLE component

### 1.3.3.5 DO DASH\_OFFSET

Specifies the position in the dash pattern where to start a dashed line. For example, if DASH\_LIST == (5,2) and DASH\_OFFSET == 5, the line starts with the gap.

Value:	Positive integer
Default value:	0

**11.3.3.6 FT FONT**

Specifies the X-windows name of the font used to draw a text.

Value:	The value can be given in the following four ways (see <a href="#">Section 11.3.4</a> ): With a font name given as a text, e.g. "KANJI_24" With a font number given as an integer expression, e.g. 3. With scope and font number given as a vector of two elements: (scope,font_number), e.g., ("M",3). As a list with one or several of the following attributes: FAMILY (FA), POINT_SIZE (PS), FACE (FC).
Default:	The MicroSCADA semi-graphical font which is called "MICROSCADA-SEMIGRAPHICS".

**11.3.3.7 FG FOREGROUND**

This component specifies the color in which the graphical element is displayed.

Values:	The color can be given in the following four ways (see <a href="#">Section 11.3.4</a> ): With a color name given as a text, e.g., "LIGHT BLUE". With a RGB number given as a vector of three elements, e.g., (10000,20000,30000). With a color number given as an integer expression, e.g. 5. With scope and color number given as a vector of two elements: (scope,color_number), e.g., ("M",5).
Default value:	"WHITE"

**11.3.3.8 FU FUNCTION**

This component states how the color (RGB values) of added pixels ('new') is combined with the color of already existing pixels ('old') on screen by means of logical operators.

Value:	"CLEAR"	0
	"AND"	new AND old
	"ANDREVERSE"	new AND (NOT old)
	"COPY"	new
	"ANDINVERTED"	(NOT new) AND old
	"NOOP"	old
	"XOR"	new XOR old
	"OR"	new OR old
	"NOR"	(NOT new) AND (NOT old)
	"EQUIV"	(NOT new) XOR old)
	"INVERT"	(NOT old)
	"ORREVERSE"	new OR (NOT old)
	"COPYINVERTED"	(NOT new)

Table continues on next page

	"ORINVERTED"	(NOT new) OR old
	"NAND"	(NOT new) OR (NOT old)
	"SET"	1
	"SCIL_XOR"	Proprietary implementation of XOR, preserved for compatibility
Default value:	"COPY"	

Examples:

"COPY" means that the added color replaces the former color.

Using "SCIL\_XOR" or "XOR" repeatedly on the same graphic element alternately draws and deletes the graphics.

### 11.3.3.9 JS JOIN\_STYLE

Specifies how corners are drawn for wide lines drawn with a single graphics command, see [Figure 23](#).

Values:	"MITER"	The outer edges of the two lines are extended to meet in one point.
	"ROUND"	The lines are joined by a circular arc with $r = \text{half the line width}$ centered on the join point.
	"BEVEL"	The outer edges of the lines are joined at the end points.
Default:	"MITER"	

### 11.3.3.10 LS LINE\_STYLE

Specifies how the line is drawn and in which colors.

Values:	"SOLID"	The line is continuous (without dashes) and drawn in the FOREGROUND color.
	"ONOFFDASH"	The line is dashed with the dashes drawn in the foreground color.
	"DOUBLEDASH"	The line is dashed with the dashes drawn in the foreground color and the gaps between the dashes in the background color.
Default value:	"SOLID"	

### 11.3.3.11 LW LINE\_WIDTH

Specifies the width of the line in coordinate units (see [Section 11.4](#)). Wide lines are drawn centered between the start and the end point given in the drawing command, see [Figure 21](#).

Values:	Integer or real, $\geq 0$ .
Default value:	0



LINE\_WIDTH=0 specifies the width to one pixel regardless of the scaling. LW=0 is the most efficient line width and fastest to draw.

**11.3.3.12 NA NAME**

A freely chosen name of the graphics context. This component is not copied when copying a context.

Value: Text.

**11.3.4 Colors and fonts****11.3.4.1 Colors**

The final color of a graphical element displayed on screen depends on the FOREGROUND, BACKGROUND and FUNCTION components of the graphics context ([Section 11.3.3](#)). The colors can be defined in four manners:

- By color names. The names of the colors in the semi-graphic pictures are: WHITE, BLACK, RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW.
- By RGB intensities given as a vector of three elements where the first element defines the red, the second element the green and the third element the blue intensity of the color as an integer or real value in the range 0 ... 65 535. Consequently, for example, (0,0,0) is black and (65535,65535,65535) is white.
- By a color mix number. A color mix number is a given small integer used to identify a color definition. The number is defined by the .COLOR command, see below.
- With scope and color mix number given as a vector of two elements: (scope,color\_number), e.g., ("M",5).



If the operating system is not capable to provide the requested color exactly (e.g. because the palette of the server is full due to the use of many different colors on screen), the nearest possible color is used.

The .COLOR command below is used for creating and modifying color mix numbers, and the COLOR function for reading the RGB values of colors.

**11.3.4.2 .COLOR [scope,] number : color [,SHARED]**

The .COLOR command defines a color mix number.

'scope'	The scope of the color number. The scope is given in the same way as for graphics contexts in <a href="#">Section 11.3.2</a> .
'number'	Color number given as a positive integer expression. Each scope can have the following maximal number of colors: "C": 20, "R": 20, "M": 50, "U": 50.
'color'	Color definition given in one of the four manners described above.

The color mix number can be used for color selection in the graphics contexts (BACKGROUND and FOREGROUND).

The optional argument SHARED has been preserved for compatibility reasons. It has no effect in current implementation.

**11.3.4.3 COLOR([scope,]number)**

Returns the RGB values of the color specified by the arguments.

'scope'	The scope of the color
'number'	The number of the color
Value:	Vector of three elements

Example:

```
#local RGB
.COLOR 1: "NAVYBLUE"
.GC : FG = 1
.BOX 100, 100, 100, 100, FILL
RGB = COLOR(1)
RGB(1) = RGB(1) + 10000
.COLOR 1 : RGB
.BOX 200, 100, 100, 100, FILL
```

The example creates a color mix initialized as navy blue, then draws a filled box in the selected color. More red is added to the color and another box is drawn adjacent to the first one.

#### 11.3.4.4 Fonts

The font used in texts is specified by the FONT component of the graphics context ([Section 11.3.3](#)). Fonts can be given as follows:

- With a font name given as a text, e.g. "KANJI\_24".
- With a font number given as an integer expression, e.g. 3. The font numbers are defined by the .FONT command, see below.
- With scope and font number given as a vector of two elements: (scope, font\_number), e.g., ("M", 3).
- As a list with one or several of the following attributes: FAMILY (FA), POINT\_SIZE (PS), FACE (FC).

The .FONT command below defines font numbers, and the FONT function is used for reading font numbers.

#### 11.3.4.5 .FONT [scope,] number : font

Defines a font number.

'scope'	The scope of the font number. The scope is given in the same way as for graphics contexts in <a href="#">Section 11.3.2</a>
'number'	Font number given as a positive integer expression. Each scope can have the following maximal number of fonts: "C": 10, "R": 10, "P": 10, "M": 20, "U": 20. 0 = the MicroSCADA semi-graphic font. Do not change it!
'font'	Font definition given in any of the four ways allowed for the FONT component, see <a href="#">Section 11.3.3</a> .

#### 11.3.4.6 FONT([scope,] number)

Returns the attributes of a font.

'scope'	The scope of the font.
'number'	The number of the font.
Value:	The font function returns a list with the attributes shown in <a href="#">Figure 24</a> . Each attribute has a two-letter alias name.

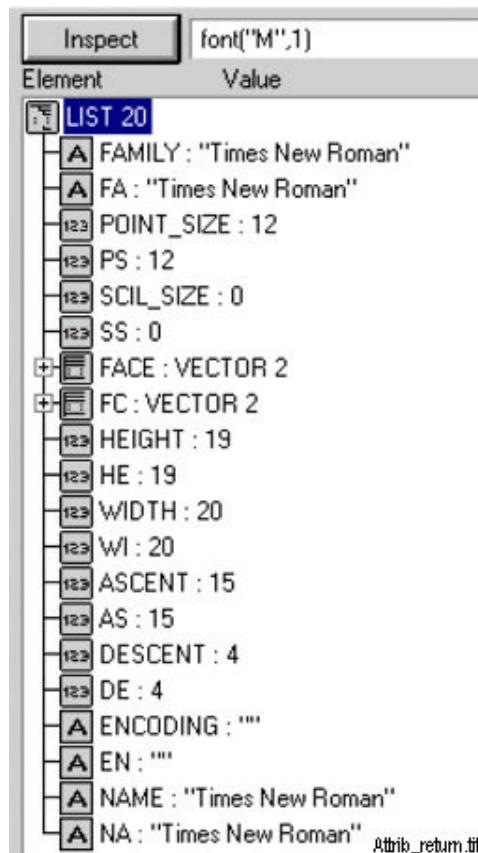


Figure 24: The attributes returned by the font function

NAME, NA	Name
HEIGHT, HE	Height (number of pixels)
ASCENT, AS	Ascent (logical extent above baseline in number of pixels)
DESCENT, DE	Descent (logical descent below baseline in number of pixels)
WIDTH, WI	Width (width of the widest character in the font in number of pixels)
FAMILY, FA	Family (the name of the font family)
POINT_SIZE, PS	Point size
SCIL_SIZE, SS	SCIL size
FACE, FC	Face
ENCODING, EN	Encoding

Note that HEIGHT is always the sum of ASCENT and DESCENT.

### 11.3.5 Reading graphics contexts

The defined graphics contexts can be read with the following SCIL functions:

#### 11.3.5.1 GC([scope,]n)

The function returns the values of the components of the context.

'scope'	The scope of the context, see <a href="#">Section 11.3</a> .
'n'	Graphics context number, integer 0 ... 20 or 0 ... 50 depending on the scope, see <a href="#">Section 11.3.2</a> .
Value:	List, where the abbreviated names of the components are the attributes.

Example:

```
@GC_DEF = GC(0)
!SHOW INFO "CURRENT DEFAULT FONT IS " + GC_DEF:VFT
```

### 11.3.5.2 COLOR\_IN([scope,] n)

The function returns the RGB values of the foreground color in the context as a vector of three elements.

'scope'	Scope of the context.
'n'	Graphics context number.

### 11.3.5.3 FONT\_IN([scope,] n)

The function returns a list value with the same attributes as for the FONT function, see [Section 11.3.4](#).

'scope'	Scope of the context.
'n'	Graphical context number.

## 11.4 Graphics canvas

### 11.4.1 General description

The graphical elements are displayed on the selected canvas which can be:

- The object (picture, picture function, dialog object) which contains the command (default).
- The root object - the main picture or the main dialog.
- A named window or dialog object.
- The parent object of the object where the command is executed.

The canvas can be selected with the commands in next section Selecting Canvas (together with the context definitions and the scaling factor) can be temporarily stored. See [Section 11.5](#).

### 11.4.2 Selecting canvas

If not changed with any of the commands listed below, the canvas for the graphics commands is the object where the commands are executed. Use the following commands in order to get the elements displayed on another canvas:

### 11.4.3 .CANVAS object

The command selects the named picture object (window picture or picture function) or dialog object (dialog or dialog object) - for canvas.

'object' A picture reference or a Visual SCIL object reference, see [Section 6](#).

## **11.4.4 .CANVAS ROOT**

The root object is used as canvas. In a picture the root object is the main picture. In a dialog system, the root is the main dialog or picture container.

## **11.4.5 .CANVAS PARENT**

The parent object is used as canvas. In a picture the parent object is the parent picture of the current window picture or picture function. In a dialog system, the parent is the parent object of the current object.

## **11.4.6 .CANVAS CURRENT**

The current object is used as canvas, i.e., the object containing the graphics command. Hence, this command returns the canvas to the default.

#### **11.4.7 .COORDINATE SYSTEM coordinate system**

Specifies the coordinate system to be used in graphic drawing commands and mouse handling commands.

The coordinate system is specified by one of the following key words: SCIL, VS.

Changes of the coordinate system with the COORDINATE\_SYSTEM command are valid only within the SCIL program where the change was made.

### Example:

## .COORDINATE SYSTEM SCIL

## 11.4.8 The SCIL coordinate system

Within the SCIL coordinate system, the (0,0) coordinate lies in the upper left corner of the canvas. The relation between SCIL coordinates and pixels on screen depends on the semigraphic font used in the SYS600 Monitor and on the scaling factor used by the drawing commands. See [Figure 25](#).

`pixel_coord = SCIL_coord *MicroSCADA_Monitor_Width / scaling_factor`

where

**pixel coord** = the pixel coordinates as displayed on screen.

**SCIL\_coord** = the coordinates given in the graphics SCIL command.

MicroSCADA Monitor Width = 80 \* Width of semigraphic character

= 80\*8 or 80\*12 or 80\*15

scaling factor

= the scaling factor. The default scaling factor is 1280. The scaling can be changed with the .SCALING command see [Section 11.4.11](#).

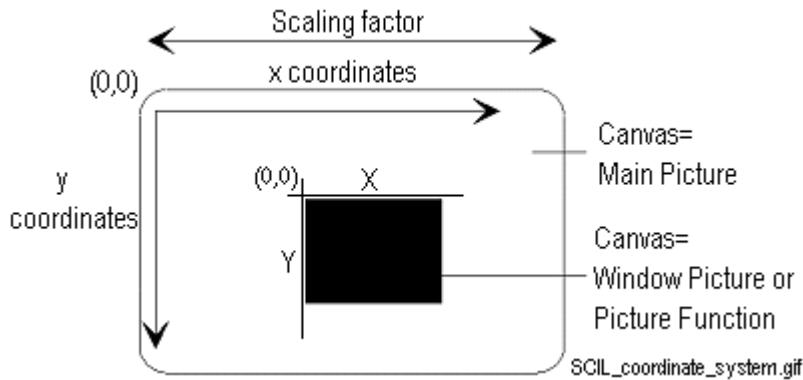


Figure 25: The SCIL coordinate system

### 11.4.9 The Visual SCIL coordinate system

Within the Visual SCIL coordinate system, the (0,0) coordinate is situated in the lower left corner of the canvas. The Visual SCIL coordinates are equal to the pixel positions within the canvas.

When handling dialogs, the origin of the coordinate system lies at the bottom left corner of the screen. When handling dialog items, the origin is in the lower left corner of the parent object. A unit in the coordinate system is a pixel.

Coordinates are also used, for example, when giving a position with the \_GEOMETRY attribute and when reading a position with the \_GET\_POINTER\_POS method. For more information on VS coordinates, see Section 2.4 in the Visual SCIL manual.

The Graphics Contexts components LW, DL, DO are defined according to the active coordinate system. These Graphics Contexts components are not affected by the coordinate system used at drawing.

Example:

```
.COORDINATE_SYSTEM VS
.GC : LW = 5
.LINE 10,10,100,100 ;line width is now 5 units according to
;the VS coordinate system (pixels).
.COORDINATE_SYSTEM SCIL
.LINE 10,10,100,100 ;line width is still 5 units according to
;the VS coordinate system (pixels),
;because the Graphics Components were defined under
;VS coordinate system
```

### 11.4.10 Changing Scaling Factor

The scaling factor can be changed by the following command:

### 11.4.11 .SCALING [s]

Sets the scaling factor to be used in the subsequent SCIL graphics commands.

's'	is an integer or real expression, the scaling factor. If 's' = 0, the coordinates are taken as pixel coordinates of the screen where the graphical element is displayed. If 's' is omitted, the scaling factor is returned to the default (= 1 280).
-----	--

For example, a graphical element programmed with  $s = 0$  will have different sizes depending on the resolution of the screen where it is shown.

The scaling factor can be temporarily stored (together with context definitions and canvas selections) by the commands in [Section 11.5](#). Note that the scaling factor only affects the SCIL coordinate system.

## 11.4.12 Mouse input

The coordinates can be read from the cursor position with the .MOUSE command. The .MOUSE command can be used with or without tracing. If tracing is OFF, the command reads the cursor position and the current mouse button states. If tracing is on, the command notes the following mouse events: button press and release, movement of mouse (provided that motion event is on, see the .MOUSE ON command). By means of the .MOUSE ON and .MOUSE OFF commands, tracing is switched on and off (default = OFF).

The .MOUSE DISCARD command is used to discard the pending mouse clicks.

## 11.4.13 .MOUSE x, y [, button [, buttons [, RELATIVE] ] ]

Reads the SCIL coordinates of the cursor and the mouse button states.

'x' and 'y'	Two variables that get the value of the x and y coordinates respectively (SCIL or VS coordinates).
'button'	A variable that gets the value of the pressed or released mouse button number (1, 2 or 3). If the registered mouse event was a motion event, the variable gets the value 0. If tracing is OFF, the variable gets the value 0. If a local variable by the name exists, it is used, otherwise a global variable.
'buttons'	A variable that receives the current mouse button states if tracing is OFF, or the button states immediately before the mouse event if tracing is ON. The states are returned as a bit mask, where each bit number represents a mouse button. The bit values have the following meanings: 0 = the button is released, 1 = the button is held down. If a local variable by the name exists, it is used, otherwise a global variable.
RELATIVE	An optional keyword. If RELATIVE is given as the last argument of the command, the coordinates are relative to the current canvas (If the canvas is not explicitly set, the window or picture function executing the .MOUSE command acts as the current canvas). If RELATIVE is not given, the coordinates returned by .MOUSE command are relative to the part picture (picture shown in window) executing the command.

Examples:

```
.MOUSE X, Y, RELATIVE
.MOUSE X, Y, BUTTON, RELATIVE
.MOUSE X, Y, BUTTON, BUTTON_MASK, RELATIVE
```

## 11.4.14 .MOUSE ON [MOTION], .MOUSE OFF

.MOUSE ON sets the program in tracing state. If the MOTION option is given, also the motion of the cursor is traced.

.MOUSE OFF ends the tracing state. The tracing state is automatically ended when a program is completed.

Example:

The following SCIL sequence draws a line segment from (0,0) to the position pointed by the user. The final position is given by releasing button 1.

```
.MOUSE ON
@B1_PRESSED = FALSE
#LOOP NOT %B1_PRESSED
    .MOUSE X, Y, BUTTON, BUTTONS
    #IF (%BUTTON ==1) AND (BIT(%BUTTONS , 1)==0) #THEN #BLOCK
        @B1_PRESSED = TRUE
    #BLOCK-END
#LOOP-END
```

The program sequence above waits until mouse button 1 is pressed, then the following is executed:

```
.MOUSE ON, MOTION
.GC : FUNCTION = "XOR"
.LINE 0, 0, %X, %Y
#LOOP %B1_PRESSED
    .MOUSE NEW_X, NEW_Y, BUTTON, BUTTONS
    .LINE 0, 0, %X, %Y
    #IF %BUTTON == 1 #THEN @B1_PRESSED = FALSE
    #ELSE #BLOCK
        .LINE 0, 0, %NEW_X, %NEW_Y
        @X = %NEW_X
        @Y = %NEW_Y
    #BLOCK-END
#LOOP-END
```

The line is drawn and erased until the mouse button is released, then the following is executed:

```
.GC : FUNCTION = "COPY"
.LINE 0, 0, %NEW_X, %NEW_Y
.MOUSE OFF
```

The line is drawn from (0,0) to the coordinates given be the variables %NEW\_X and %NEW\_Y.

## 11.4.15 .MOUSE DISCARD

.MOUSE DISCARD discards all the pending mouse clicks, i.e. the mouse clicks that have not yet been processed.

This command may be used after a lengthy calculation to discard the mouse clicks that are done by an impatient operator.

After a new dialog has been displayed, this command may be used to cancel any mouse clicks that may have been done before the dialog was seen.

Unlike other .MOUSE commands, .MOUSE DISCARD may be used both within a main dialog context and within a picture container context (the others work only in a picture container context).

## 11.5 Miscellaneous graphical commands

### 11.5.1 Storing and restoring selections

#### 11.5.1.1 .PUSH, .POP

The commands .PUSH and .POP are used for storing temporarily the context definitions, the scaling factor and the canvas selection. They are useful when there is a need to store standard selections while making other selections, and then restore the standard selections after a while. They are used, for example, when a subroutine uses specific values for canvas, scaling and graphics contexts, which should not interfere with the picture that executes the subroutine.

.PUSH stores the current canvas selection, scaling factor and graphics contexts definitions. .POP restores the selections stored with .PUSH. The .POP command must be located in the same program as the corresponding .PUSH command.

### 11.5.2 Display handling commands

#### 11.5.2.1 .FLUSH

This command forces a blink timing and an immediate updating of the entire application window. However, if there is an ongoing pending, the command has no effect. The command is rather time consuming.

#### 11.5.2.2 .PEND ON, .PEND OFF

Pending is used to prevent disturbing flickering when drawing related graphics primitives in sequence (for example when changing position of an object in animation).

The graphics drawn after PEND ON are not shown on screen until the matching PEND OFF is encountered. At PEND OFF the resulting output of the intervening commands is shown as a flash (that is the drawn primitives are not displayed one by one). PEND ON and the matching PEND OFF must be located in the same SCIL program. If a SCIL program ends while the output is pending, an automatic PEND OFF is generated by the base software. PEND commands may be nested. In this case, the outermost PEND OFF triggers the output.

An implicit PEND ON - PEND OFF is set by the base software around the semi-graphical background and the draw program when a picture is displayed.



# Section 12 SCIL programming guide

This section provides a programming guide for the most important SCIL tasks. The guide gives brief instructions for accomplishing various tasks with SCIL and refers to the sections in this manual and other manuals where the used SCIL elements are detailed. The following main subjects are discussed:

- Picture handling
- Visual SCIL object handling
- Program execution
- Process supervision and control
- Alarm and event handling
- Calculations and reports
- System configuration and communication
- Application database management
- Error handling

## 12.1 Picture handling

*Table 6: Loading Pictures*

Task	Use	Comments
Loading a new picture	!NEW_PIC command	<a href="#">Section 9.3</a>
Re-loading the previous picture	!LAST_PIC command	<a href="#">Section 9.4</a>
Loading an alarm picture	!INT_PIC command	<a href="#">Section 9.4</a> The alarm picture is process object specific and defined in the process object definition.

*Table 7: Window Handling*

Task	Use	Comments
Showing or updating windows	!SHOW command	<a href="#">Section 9.4</a>
Erasing windows	!ERASE command	<a href="#">Section 9.4</a>
Showing window background	!SHOW_BACK	<a href="#">Section 9.4</a>
Creating windows with SCIL	!WIN_CREATE or !WIN_NAME	<a href="#">Section 9.4</a>
Positioning windows with SCIL	!WIN_POS	<a href="#">Section 9.4</a>
Defining window expression with SCIL	!WIN_INPUT or !SHOW	<a href="#">Section 9.4</a>
Defining window representation with SCIL	!WIN_REP	<a href="#">Section 9.4</a>
Defining window picture with SCIL	!WIN_PIC	<a href="#">Section 9.4</a>
Changing window level	!WIN_LEVEL	<a href="#">Section 9.4</a>
Reading window attributes	attribute reference: {picture}.attribute	<a href="#">Section 6.4</a>
Writing window attributes	.SET + attribute reference	<a href="#">Section 9.3</a>

*Table 8: Named Programs*

Task	Use	Comments
Executing named programs	Program call: {picture}.name{(arguments)}	<a href="#">Section 6.4</a>
Using named programs in expressions	Program call as above.	<a href="#">Section 6.4</a> Possible only for named programs which return values.
Building named programs which use arguments and return a value	Use the arguments ARGUMENT functions to read the arguments given in the program call. Use the #RETURN command to return a value.	The ARGUMENT functions are described in <a href="#">Section 10</a> . The #RETURN command in <a href="#">Section 9.2</a> .

*Table 9: Updating Pictures*

Task	Use	Comments
Cyclical updating	Updating program. Start the updating and define the updating interval with the !UPDATE command.	<a href="#">Section 9.4</a> The updating program should not be comprehensive.
Event based updating	#ON blocks + event object activation	<a href="#">Section 9.2</a> Event object activation from process objects requires that the EE attribute is = 1.

*Table 10: Miscellaneous*

Task	Use	Comments
Stopping function key blinking	!RESTORE	<a href="#">Section 9.4</a>
Hardcopy (semi-graphic) of picture	!SEND_PIC	<a href="#">Section 9.4</a>
Closing application windows	!CLOSE	<a href="#">Section 9.4</a>

## 12.2 Visual SCIL object handling

*Table 11: Loading, Creating and Deleting Objects*

Task	Use	Comments
Loading objects stored in a Visual SCIL object file	.LOAD command	<a href="#">Section 9.3</a> .
Creating objects with SCIL	.CREATE command	<a href="#">Section 9.3</a> .
Deleting objects	.DELETE command	<a href="#">Section 9.3</a> .

*Table 12: Executing Methods*

Task	Use	Comments
Executing methods from SCIL	method call: {object}.method{(arguments)}	<a href="#">Section 9.3</a> . Such methods which are executable with SCIL.
Using method calls in expressions	Program call as above	<a href="#">Section 9.3</a> . Possible only for methods which return values.
Building methods which use arguments and return a value	Use the arguments ARGUMENT functions to read the arguments given in the program call. Use the #RETURN command to return a value.	The ARGUMENT functions are described in <a href="#">Section 10</a> . The #RETURN command in <a href="#">Section 9.2</a> .

*Table 13: Reading and Writing Attributes*

Task	Use	Comments
Reading attributes (functions and features)	Attribute reference: {object}.attribute{[arg:s]}	<a href="#">Section 6.4</a>
Modifying attributes	.SET command + attribute reference .MODIFY command + object reference + attribute list	<a href="#">Section 9.3</a> . Attributes can also be modified with the .LOAD and .CREATE commands.

## 12.3 Program execution

*Table 14: Executing Programs*

Task	Use	Comments
Executing command procedures	#EXEC, #EXEC_AFTER	<a href="#">Section 9.2</a> . Also started by time channels and event channels.
Executing named programs in pictures		See above.
Executing methods in Visual SCIL objects		See above.
Executing a program written as text vector, for example, in a text file	#DO command DO function Use TEXT_READ to read a file to a text vector	<a href="#">Section 9.2</a> . <a href="#">Section 10</a> <a href="#">Section 10</a>

*Table 15: Miscellaneous*

Task	Use	Comments
Conditional execution of program block	#IF ... #THEN #ELSE_IF..... #THEN #ELSE	<a href="#">Section 9.2</a> .
Building program blocks within programs	#BLOCK statements #BLOCK_END	<a href="#">Section 9.2</a> .
Executing different program blocks depending on the situation	#CASE #WHEN #OTHERWISE #CASE_END	<a href="#">Section 9.2</a> .
Executing program loops	#LOOP #LOOP_END #LOOP_WITH #LOOP_EXIT	<a href="#">Section 9.2</a> .
Declaring variables and arguments	#LOCAL #ARGUMENT	<a href="#">Section 9.2</a> .
Pausing the program execution	#PAUSE	Used in exceptional cases
User defined functions	DO function with arguments in the program: read arguments with the ARGUMENT functions. Return values with the #RETURN command.	<a href="#">Section 10</a> <a href="#">Section 10</a> <a href="#">Section 9.2</a> .
Stopping program execution, possibly returning a value	#RETURN	<a href="#">Section 9.2</a> .

*Table 16: Process Supervision and Control*

Task	Use	Comments
Using process object value in expressions	Process object notation	<a href="#">Section 6.3</a> Process objects are described in the Application Objects manual.
Controlling process objects	#SET command + process object notation	<a href="#">Section 6.3</a> and <a href="#">Section 9.2</a> .

*Table 17: Alarm and Event Handling*

Task	Use	Comments
Alarm and event handling features		Defined in the process objects, see the Application Objects manual.
Building alarm and event lists	APPLICATION_OBJECT_LIST HISTORY_DATABASE_MANAGER #INIT_QUERY PROD_QUERY	<a href="#">Section 9.2</a> . <a href="#">Section 10</a> .
Printing event and alarm information	Automatically from process object or with #PRINT command and PRINT_TRANSPARENT if full graphic printout	Defined in process object. <a href="#">Section 9.2</a> . <a href="#">Section 10</a> . Requires a printout picture.

*Table 18: Calculations and Reports*

Task	Use	Comments
Executing Command Procedures	#EXEC, #EXEC_AFTER + command procedure	<a href="#">Section 9.2</a> . Command procedures detailed in the Application Objects manual. Also: time channels, event channels.
Acquiring Report Data	#EXEC, #EXEC_AFTER + data object	<a href="#">Section 9.2</a> . Data objects detailed in the Application Objects manual. Also: time channels, event channels.
Reading and Using Stored Data	Data object notation	<a href="#">Section 6.3</a>
Editing report data	#SET + data object notation	<a href="#">Section 9.2</a> .

*Table 19: System Configuration and Communication*

Task	Use	Comments
Creating base system objects	#CREATE command + LIST function	<a href="#">Section 9.2</a> . <a href="#">Section 10</a>
Defining NET lines	#SET + NET line attribute PO	<a href="#">Section 9.2</a> . The System Objects manual May also be done in preconfiguration
Creating communication system objects	#SET + NET object and device creation attribute	<a href="#">Section 9.2</a> . The System Objects manual May also be done in preconfiguration
Setting system object attributes	#SET + object notation	<a href="#">Section 9.2</a> . The System Objects manual.
Starting PC-NET	Defining link base system object	The System Objects manual.

*Table 20: Application Database Management*

Task	Use	Comments
Creating application objects	#CREATE	<a href="#">Section 9.2. Tools</a>
Deleting application objects	#DELETE	<a href="#">Section 9.2. Tools</a>
Modifying application objects	#MODIFY	<a href="#">Section 9.2. Tools</a>
Searching among objects	#SEARCH, NEXT, PREV APPLICATION_OBJECT_LIST	<a href="#">Section 9.2. Section 10 Tools</a> <a href="#">Section 10.</a>
Copying objects	#CREATE + FETCH, PHYS_FETCH, DATA_FETCH	<a href="#">Section 9.2. Section 10 Tools</a>

*Table 21: Error Handling*

Task	Use	Comments
Error handling policy	#ERROR STOP #ERROR CONTINUE #ERROR IGNORE	<a href="#">Section 9.2.</a>
Error handling programs	In pictures: Named program named ERROR_HANDLER In Visual SCIL Objects: Error handling method.	See below. See Visual SCIL User Interface Design.
Reading and writing error status	STATUS function SET_STATUS	<a href="#">Section 10. The Status Codes manual.</a> Argument in error handling programs

### 12.3.1 Error Handling in Pictures

The error handling in pictures can be defined by a named program with the name ERROR\_HANDLER. When a SCIL error occurs in a picture program, the ERROR\_HANDLER program, if it exists, is started. The following 6 arguments are transferred to the error handler and can be used in the named program by means of the ARGUMENT functions (see [Section 10.9](#)).

- The SCIL status code (integer)
- The picture and the program where the error occurred (text) given as a picture path starting from the main picture, for example:

.UPDATE (= the update program of the main picture)

WINDOW1/PIC\_FUNC\_1/WINDOW2.MY\_NAMED\_PROGRAM

- The erroneous SCIL line (text)
- The column position within the line (integer, may be 0)
- Current error handling policy (text), either "STOP" or "CONTINUE"
- Program line number (integer)

The error handler program is searched for in the following order:

1. The picture - main picture, window picture or picture function - where the error occurred.
2. The picture functions of the picture where the error occurred.
3. The parent of the picture where the error occurred.
4. The picture functions of the parent picture.
5. The parent of the parent picture, etc., up to the main picture.

The first error handler program found is executed.

If no error handler is found, the standard picture error message is shown on the top line of the main picture. See the Status Codes manual.



# Section 13 SCIL editor

This section describes the SCIL Editor when it is accessed from the Tool Manager or from other tools like Picture Editor, Command Procedure Object Definition tool and Dialog Editor for Visual SCIL.

## 13.1 General

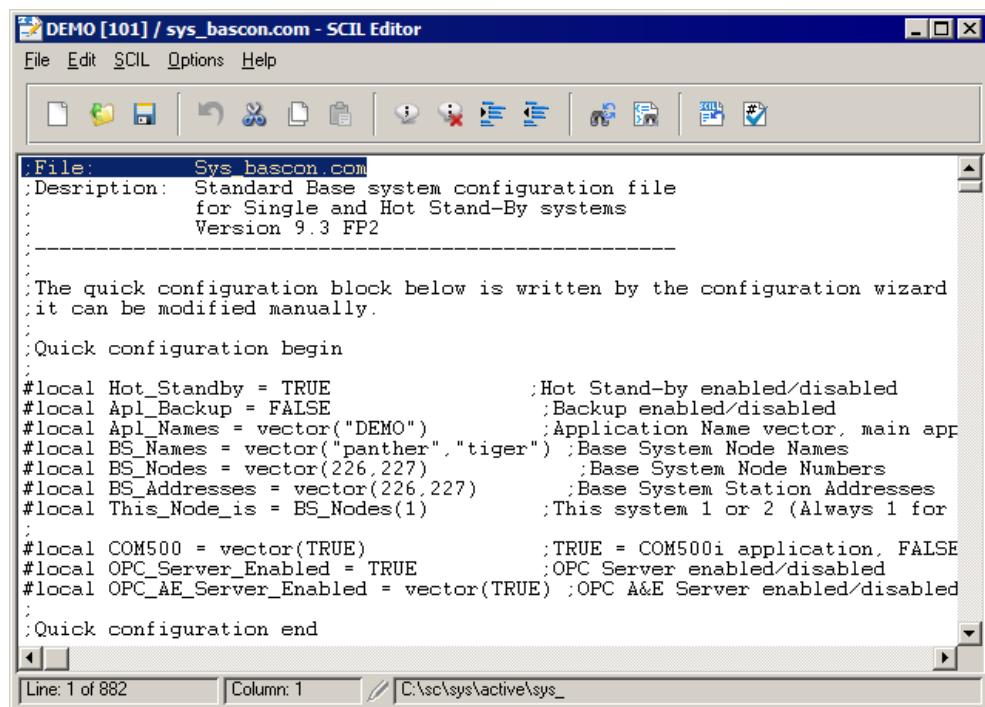


Figure 26: The SCIL editor as accessed from the Tool Manager

The SCIL editor is a text editor designed for editing text files and SCIL programs. The editor has ordinary editing functions and tools for assisting design of SCIL program code. The assistant tools for SCIL programming are dialogs for inserting SCIL commands, statements and functions as well as syntax checking. [Figure 26](#) shows the SCIL editor as opened from the Tool Manager. The menu bar and the toolbar are located above the text window and below there is a status bar. The status bar shows the current cursor position in terms of rows and columns. The active path is also shown in a field on the status bar. Unsaved changes in the program are indicated with a colored pencil on the status bar. The pencil is dimmed when there are no unsaved changes.

## 13.2 Menus

Commands of the **File** menu:

<b>New</b>	Open a new file with the default name ‘Untitled.txt’. In case there are unsaved changes to the current file a dialog asking if changes are to be saved is shown.
<b>Open...</b>	Open an existing file. Clicking <b>Open</b> displays a File Chooser dialog box for selecting the file to open.
<b>Save</b>	Save the file without changing the filename. The <b>Save</b> command acts as <b>Save As</b> if invoked on a new file.
<b>Save As...</b>	Save the file. A File Chooser dialog box for choosing path and file name is opened.
<b>Import...</b>	Insert the contents of a text file after the current line. Opens a File Chooser dialog box for selecting the file to be imported.
<b>Export...</b>	Export the whole text or the selected text. Opens a File Chooser dialog box, with a default file name EXPORTED.TXT, for selecting the file to export to.
<b>Print Setup...</b>	This command opens a standard Print Setup dialog provided by the Windows operating system.
<b>Print...</b>	This command opens a standard Print dialog provided by the Windows operating system.
<b>File history commands</b>	The names and the paths of the most recently opened files. The maximum number of the file names and paths shown can be determined from the <b>File history commands</b> in the <b>Options</b> menu.
<b>More History</b>	This submenu is placed under the 5th file history command and displays the possible commands from 6 to 20. The submenu is visible if <b>File history commands</b> is set to more than 5, and if more than 5 files have been opened after the setting was made. The commands on the submenu do not contain any mnemonics.
<b>Exit</b>	Exit the SCIL editor. The user is informed if there are unsaved changes to the text.

The commands of the **File** menu differ whether the SCIL editor is opened from the Tool Manager or from within another tool. When the SCIL editor is opened from the Tool Manager the **File** menu contains the commands **New/Open/Save** and **Save As...**, while opened from within another tool these commands are replaced by the **Update** command.

#### Commands of the **Edit** menu:

<b>Undo</b>	Undo last executed command.
<b>Redo</b>	Redo last executed command.
<b>Cut</b>	Cut selected text and place on the Clipboard.
<b>Copy</b>	Copy selected text and place on the Clipboard.
<b>Paste</b>	Paste the text of the Clipboard at current cursor position.
<b>Clear</b>	Clear selected text. The cleared text is not placed on the Clipboard.
<b>Select All</b>	Select the whole text.
<b>Comment</b>	Insert a semicolon at the beginning of the current or selected line(s). The semicolon acts as sign for commenting and the SCIL interpreter recognises lines starting with a semicolon as a remark.
<b>Uncomment</b>	Delete the semicolon from the beginning of current or selected line(s).
<b>Indent</b>	Increase indent of current or selected line(s).
<b>Unindent</b>	Decrease indent of current or selected line(s).
<b>Modify All Lines/Upper Case</b>	Converts the lines to upper case letters.
<b>Modify All Lines/Lower Case</b>	Converts the lines to lower case letters.

Table continues on next page

<b>Modify All Lines/Capitalize</b>	Converts the first character to upper case and the rest to lower case letters.
<b>Modify All Lines/Left Trim</b>	Removes leading spaces from the lines
<b>Modify All Lines/Right Trim</b>	Removes trailing spaces from the lines
<b>Modify All Lines/Compress Spaces</b>	Replaces multiple spaces with a single space.
<b>Modify All Lines Sort Alphabetically</b>	Sorts the lines in the alphabetical order.
<b>Modify All Lines/Sort by Length</b>	Sorts the lines in the length order.
<b>Modify All Lines/Reverse Order</b>	Reverses the order of the lines.
<b>Modify All Lines/Remove Duplicates</b>	Removes duplicate lines.
<b>Find/Replace...</b>	Open a <b>Find/Replace</b> dialog with a field for text to find and a field for the optional replacement text.
<b>Find Next</b>	Performs the last specified find operation from the current cursor position.
<b>Find Block</b>	Find subsequent blocks of code in a SCIL program. For example blocks delimited by a loop command.
<b>Go To Line</b>	Move cursor to a line which line number is given in the dialog shown by the command.

**Commands of the SCIL menu:**

<b>Naming Standards</b>	Open a dialog showing naming conventions for SCIL variables and Visual SCIL objects / text identifiers.
<b>Insert SCIL...</b>	Open a dialog for inserting SCIL commands, functions and objects. The dialog shows the different categories in a tree structure. To expand a category click the plus sign in front of the category name. Clicking an item in the tree structure shows the syntax of the SCIL code in the text field to the right in the dialog. Clicking the <b>Insert</b> button inserts the code at current cursor position. Click <b>Close</b> to exit the dialog.
<b>Check Syntax</b>	Check the syntax of SCIL code. Information of no found errors is shown as a message. A found error is shown in a dialog along with an error code and the line number of the invalid program code. The line containing invalid syntax is shown in the text field. Edit the program and click the <b>Check</b> button to verify the correction. If the correction pass the syntax check, the next error is located and shown. When no more errors are found, a <b>Syntax Checking Successful</b> message is displayed.
<b>Status Codes...</b>	Show <b>Status Code</b> dialog. The inserted status code is shown as the mnemonic status message.

**Commands of the Options menu:**

<b>Toolbar</b>	Toolbar visible/invisible option. Checked means toolbar is visible.
<b>Assistant View</b>	Open or update read-only secondary window with a copy of the current program.
<b>Slice View</b>	Makes possible to show slices from a program. The item is enabled, when the program has more than 10000 lines.
<b>Status Updating</b>	On/off option for status bar field, menu bar items and toolbar buttons. Checked means that menu items and toolbar buttons are enabled and the status bar fields are updated according to the current status of the program.
<b>Check Syntax at Save</b>	Toggle item for turning on or off the automatic syntax checking of the program during program saving. Default state is off.

Table continues on next page

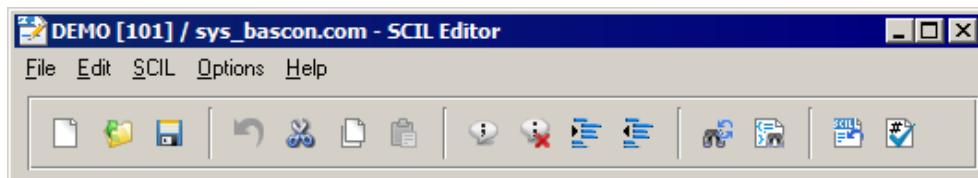
<b>Compilation In Use</b>	Toggle item for turning on or off the auto-compilation of the program during program saving. Turning off also deletes the possible existing compiled program. Enabled only if the calling tool supports program compilation. Includes syntax checking.
<b>File History Length</b>	Determines the maximum number of the names and the paths, of the most recently opened files, displayed as file history commands under the <b>File</b> menu. The file history length can be set between 0 and 20. Giving the length 0 means that the file history is disabled.
<b>Indent...</b>	Open a dialog for setting indent character count and the state for indenting new lines.
<b>Font Set Font</b>	Brings up a Font Chooser for selecting the editor font. The selected font is saved in the parameter file, and restored when the editor is opened next time.
<b>Font/Enlarge Size</b>	Find the next larger point size and set the editor font.
<b>Font/Reduce Size</b>	Find the next smaller point size and set the editor font.
<b>Font/Reset Font</b>	Set the editor font to the default value: Courier Medium Modern 10.

Commands of the **Help** menu:

<b>Shortcut Keys</b>	Shows a dialog with the sequences of keystrokes that corresponds to certain actions not included in the menus.
<b>User Parameter Saving</b>	Shows information about whether tool or user specific parameters are used to save the properties and geometry of the SCIL editor.
<b>About</b>	Shows a dialog with Tool information and System information.

### 13.3 Toolbar

The toolbar of the SCIL editor is a collection of buttons corresponding to commands found in the menus. The toolbar is shown in [Figure 27](#). The corresponding command of each button is explained in [Table 22](#). The last button **Evaluate in test dialog** (in [Table 22](#)) is present on the toolbar when the editor is opened from the **Test Dialog**.



*Figure 27: Toolbar of the SCIL editor as opened from the Tool Manager*

*Table 22: Corresponding commands of the buttons on the toolbar*

	<b>File Exit</b>		<b>Edit Copy</b>		<b>Edit Find/Replace</b>
	<b>File New</b>		<b>Edit Paste</b>		<b>Edit Find Next</b>
	<b>File Open</b>		<b>Edit Comment</b>		<b>Edit Find Block</b>

	File Save		Edit Uncomment		SCIL Insert SCIL...
	Edit Undo		Edit Indent		SCIL Check Syntax
	Edit Cut		Edit Unindent		Evaluate in test dialog

## 13.4 Opening and closing the SCIL editor

### 13.4.1 Opening the SCIL editor

The SCIL Editor is opened when you select a program for editing in the Picture Editor, Command Procedure object definition tool, **Test** Dialog or the Dialog Editor. The programs and the procedures for starting the program editing are described in the manuals Picture Editing, Visual SCIL User Interface Design and Application Objects.

The SCIL program editor can also be opened from the Tool Manager by clicking the SCIL Editor icon in the **Miscellaneous** page.

When the SCIL Editor is opened from a tool, the user already chosen a program for editing. If it exists, its contents is shown in the editor. If it is new, the editor is empty.

### 13.4.2 Opening files

After having opened the editor from the Tool Manager, the user can open a text file for editing or create a new file. This possibility is usually not available when the editor has been opened from an object tool. To open a file:

1. Click **Open** from the **File** menu. A file chooser dialog box appears.
2. Select directory from the directory tree. All files are listed as default. Four different path selection modes are supported, as described below. The default is SYS600 Relative Paths. The file list can be viewed as a list or with details by clicking on either the List () or the Details () button on the right, above the file list box.
3. Click the name of the text file, or type the file name in the data entry field below the file list. A file can be opened also as a read-only file by selecting the **Read-only** check box.
4. Click **Open**.

To close a file and start editing of another file, open another file or click **New** from the **File** menu to open a new file. If the previous file was not saved, the user is asked to save the changes or abandon them.

In the File Chooser the paths can be selected in four different modes:

Application Relative Paths	Path representation in the SYS600 path format relative to the current SYS600 application home directory. The application home directory itself can't be referenced.
SYS600 Relative Paths	Path representation in the SYS600 path format relative to the SYS600 root directory. The SYS600 root directory itself can't be referenced.
Logical Paths	Path representation in the SYS600 logical path format.
Operating System Paths	Path representation in the format used by the operating system.

### 13.4.3 Creating files

File name is given when the file is saved. If the file does not exist, a new file is created.

## 13.4.4 Saving files

The work can be saved anytime. To save a file:

1. Click **Save** or **Save As** from the **File** menu. The **Save** command saves the file with the same name, if it already exists. Use the **Save As** command to save the file with another name.
2. When selecting **Save** for the first time or **Save As**, the file chooser dialog box appears. Select the correct folder from the directory tree and click on the file name, or type it in the **Save as** text box. Four different path selection modes are supported, as described above. A new folder can also be created by clicking on the **Create New Folder** button ( ) above the file list.
3. Click **Save**.

If the SCIL Editor has been opened from an object tool, save the program by choosing **Update** on the **File** menu.

## 13.4.5 Undo operation

To undo the last editing operation, click **Undo** on the **Edit** menu. The undo operation revokes the last editing operation. The maximum number of actions that can be undone is 50.

## 13.4.6 Closing the SCIL editor

To close the editor, click **Exit** on the **File** menu.

If changes have been made since the last time the program was saved, a dialog box appears prompting to save changes.

# 13.5 Typing and editing programs and texts

## 13.5.1 Typing

The basic function of the SCIL Editor compares to common text editing programs. Most keyboard keys have their natural functions. The functions of some important keys are explained below:

Insert	toggles between insert and overwrite.
Home	moves the cursor to the beginning of the line and the End key to the end of line.
CTRL+Home	moves the cursor to the beginning of the program.
CTRL+End	moves the cursor to the end of the program or text.
<- and ->	moves the cursor one step to the left and right respectively.
CTRL+<- and CTRL+->	moves the cursor to the beginning of the next/previous word.
TAB	inserts a specified number of spaces defined by the user and moves the cursor the same number of steps to the right.

More information on shortcut keys is found by choosing **Shortcut Keys** from the **Help** menu.

## 13.5.2 Scroll feature

A lengthy program/content in the SCIL editor can be scrolled up/down by scrolling the middle mouse button.

### 13.5.3 Selecting text for editing

The whole program or parts of it can be selected for moving, copying and deleting as follows:

- To select a word, double-click it.
- To select a text section, place the cursor at the beginning of the section, press the mouse button and hold it down while dragging the cursor to the end of the text.
- To select the whole program or text, press CTRL+A or choose **Select All** from the **Edit** menu.

The selected text is shown in reversed colors. It can be moved, copied and deleted as described below. Text strings can also be replaced in the selected section.

### 13.5.4 Copying

To copy a text within the program or from one program to another:

1. Select the text you want to copy as described above.
2. Choose **Copy** from the **Edit** menu or press CTRL+C. The text is copied to the clipboard.

To move text from one program to another, activate the program to be copied.

1. Place the cursor at the position where the copied text should be inserted.
2. Click **Paste** from the **Edit** menu or press CTRL+V.

### 13.5.5 Moving text

To move text:

1. Select the text to be moved as described above.
2. Choose **Cut** from the **Edit** menu or press CTRL+X. The selection is removed from the screen and placed in the clipboard.
3. To move text from one program to another, activate the program to which the text will be moved.
4. Click the place where you want to insert the text.
5. Choose **Paste** from the **Edit** menu or press CTRL+V.

### 13.5.6 Deleting

To delete text:

1. Select the text wanted as described above.
2. Click **Clear** from the **Edit** menu or press DELETE.

The selected text disappears.

### 13.5.7 Commenting

A comment in a SCIL program is a line or a part of a line marked by a comment mark (;) at the beginning. When this sign appears in a program line, the rest of the line is regarded as a comment and not executed. Comments can be used, for example, to explain how the program works or to prevent the execution of a program line without deleting it permanently. The comment signs may be inserted and deleted using the ordinary text editing functions. To mark several subsequent lines as comments, the **Comment** and **Uncomment** commands of the **Edit** menu can also be used.

To mark a program section as comments:

1. Select the lines to be marked as comments, see above.
2. Click **Comment** on the **Edit** menu.

A semicolon is inserted as the first character of each line in the selection.

To remove the comment marks located in the beginning of lines:

1. Select the lines from which the comment marks should be removed.
2. Click **Uncomment** on the **Edit** menu.

All the comment marks that are located at the beginning of the lines are removed. Semicolons located elsewhere are not removed.

## 13.5.8 Indenting

Text in paragraphs usually extends from the left margin to the right margin. A paragraph can be indented to set it off from other text. Indenting is used to increase readability of program code. Setting the measurement for indenting is done by choosing **Indent...** from the **Options** menu. The checkbox Auto-indent enabled means that a new line is indented according to previous line.

To indent a section:

1. Select the lines to be indented. If no text is selected, the current line is indented.
2. Click **Indent** from the **Edit** Menu.

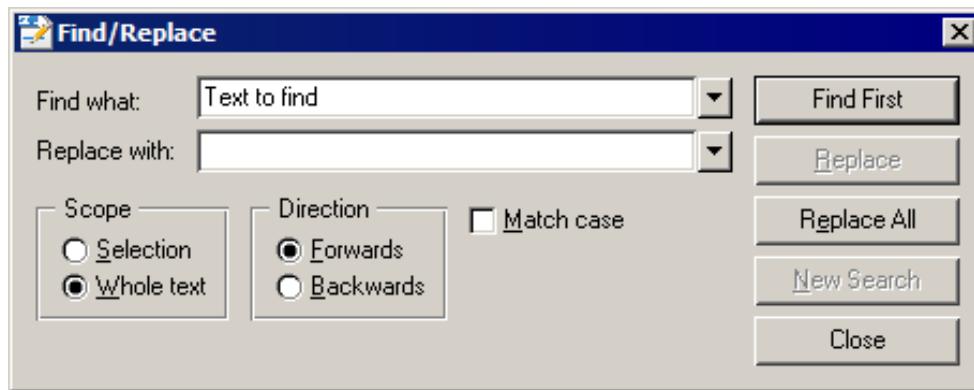
To unindent a section:

1. Select the lines to be unindented.
2. Click **Unindent** from the **Edit** Menu.

## 13.5.9 Finding text

The **Find/Replace** command searches for a given text in the program. It stops when it finds the first match and shows it as a selection. To use the **Find/Replace** command:

1. To search a certain part of the text, select the text to be searched. Otherwise the entire document is searched. The **Find or Replace** operation is always started from the current cursor position.
2. Click **Find/Replace** from the **Edit** menu. The dialog box shown in [Figure 28](#) appears. Moving the cursor is possible in the main window while the **Find/Replace** dialog box is open.



*Figure 28: You can search for text that is located somewhere in the same program using the **Find/Replace** command on the **Edit** menu*

3. In the **Find what** field, type the text to be searched for. A document can be searched either in an upward or downward direction of the program by selecting **Forwards** or **Backwards** under **Direction** in the dialog. If the case of the text (uppercase/lowercase) is of importance, select the check box **Match Case**. If a text was selected in step 1, **Selection** is selected under **Scope**.
4. Click **Find First**. If a matching text is found, the first match is shown selected in the text window. If no matching text is found, a dialog box saying “**Text not found**” appears. After the first find operation the **Find First** command button is replaced with **Find Next** command button, which can be used for finding the next occurrence. **Find Next** may be invoked repeatedly to search for the string until **Close** is clicked.

The **Find Next** operation in the main window is not possible while the **Find/Replace** dialog is open.

The last twenty items of the **Find/Replace** word lists are saved and restored when closing and opening the SCIL Editor. The lists are sorted in the chronological order, last used words first.

### 13.5.10 Replacing text

To replace the occurrences of a text string with another one:

1. To replace the occurrences found in a certain text section, select the section. Otherwise the entire document is considered. The **Find or Replace** operation is always started from the current cursor position.
2. Click **Find/Replace** from the **Edit** menu. The dialog shown in [Figure 29](#) appears. Moving the cursor is possible in the main window while the **Find/Replace** dialog box is open.

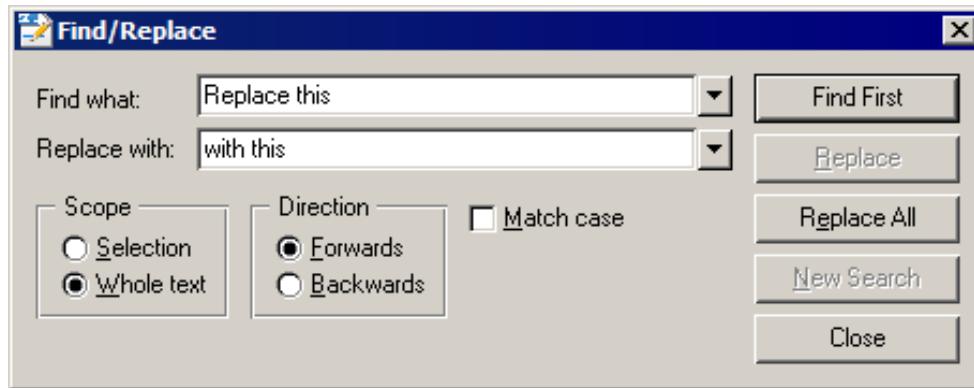


Figure 29: With this dialog you can replace one text with another

3. In the first text box, type the text you want to replace, and in the second box, type the text to be replaced it with. The user can also select whether the entire document or just the selected text section is replace. If the case of the text is of importance, select **Match Case**. Select search direction under **Direction**.
4. Click **Find First** to find the first occurrence of the text string without replacing it immediately. After the first find operation the **Find First** command button is replaced with **Find Next** command button, which can be used for finding the next occurrence. **Find Next** may be invoked repeatedly to search for the string until **Close** is clicked.
5. When an occurrence to be replaced is found, click **Replace**. This replaces the selected text and searches the next occurrence of the text string. If the found text is edited manually in the main window during a **Find or a Replace** operation, clicking **Replace** button replaces nothing, but the Find Next operation is performed. **Replace All** replaces all occurrences of the text string.

The Find Next operation in the main window is not possible while the **Find/Replace** dialog is open

The last twenty items of the Find/Replace word lists are saved and restored when closing and opening the SCIL Editor. The lists are sorted in the chronological order, last used words first.

### 13.5.11 Finding blocks

To go to a certain SCIL block in the program, click **Find Block** in the **Edit** menu. The **Find Block** command searches for the following Block commands downwards in the program starting from the cursor:

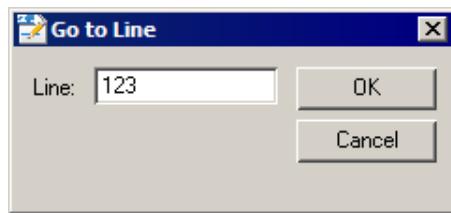
```
#BLOCK .... #BLOCK_END  
#LOOP .... #LOOP_END  
#CASE .... #CASE-END
```

When a block is found, it is selected in the text window. To find another block, place the cursor after the command that ends the previous block and then click **Find Block** again.

### 13.5.12 Finding a line

To move quickly to a certain line number:

1. Click **Go To Line** from the **Edit** menu or press **CTRL+L** on the keyboard. The dialog shown in [Figure 30](#) appears.



*Figure 30: A certain line can be moved using this dialog.*

2. In the text box, type the number of the line to which you want to move and click **OK**.

If the line number is invalid, the cursor is moved to the first line and if the given line number is too big the cursor is moved to the last line.

### 13.5.13 Importing and exporting text

Importing text means that the contents of a text file is inserted after the current line except for the case when the cursor is located at the beginning of the text, then the text is inserted at the beginning. To import a text file:

1. Click **Import...** from the **File** menu. The file chooser dialog box appears.
2. Select the correct folder from the directory tree and click on the name of the file.
3. Click **OK**.

Exporting a file means that the selected or the whole text is stored in a file. To export a text:

1. Select a text if only part of the text is to be exported.
2. Click **Export** from the **File** menu. The file chooser dialog box appears with the default file name EXPORTED.EXE in the **File name** text box.
3. Select the correct folder from the directory tree and enter a name for the file in the **File name** text box. A new folder can also be created by clicking on the **Create New Folder** button (📁) above the file list.
4. Click **OK**.

If there is an existing file with the same name, the user is asked to confirm overwriting of the existing file.

In the file chooser four different path selection modes are supported, as described in [Section 10.17](#).

### 13.5.14 Undoing and redoing operations

Most editing operations can be cancelled using the **Undo** command, for example **Cut/Copy/Paste** and **Replace** commands. When the user chooses to undo typing, undo will affect all that was written since the last editing operation, for example **Copy/Undo/Save**.

To undo an operation, click **Undo** from the **Edit** menu.

The operations that have been cancelled by using **Undo**, can be done again. This means that an **Undo** operation can also be canceled. To do this, click **Redo** from the **Edit** menu.

### 13.5.15 Insert SCIL commands, functions and objects

This assisting tool in the SCIL editor is designed to help writing SCIL code. The dialog below appears as the **Insert SCIL...** command on the **SCIL** menu is invoked. The text box to the left in the dialog contains the different categories of commands, functions and objects. Expand the nodes in the tree by clicking the plus sign. Subcategories appear as leafs in the tree structure.

Select a subcategory by clicking it. When this is done commands, functions or object definition attributes appear in the text box to the right in the dialog. The desired command, function or attribute is selected by clicking. The status bar at the bottom of the dialog displays a short description of the selected command, function or attribute.

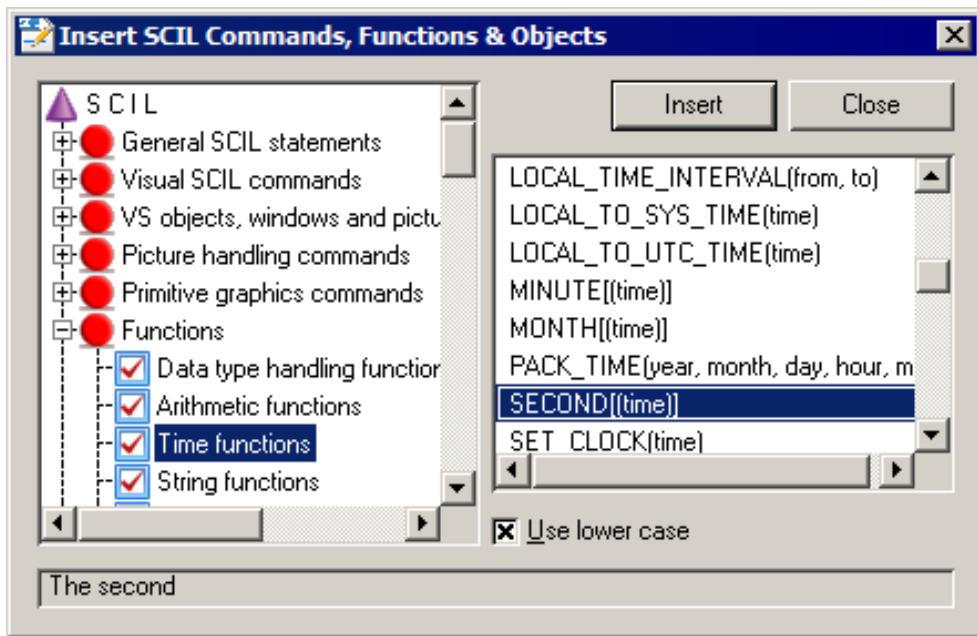


Figure 31: The Insert SCIL Commands, Functions & Objects dialog of the SCIL editor

To insert a SCIL command:

1. Place the cursor where the command is to be inserted.
2. Click **Insert SCIL...** on the **SCIL** menu.
3. Expand one of the categories by clicking the plus sign in front.
4. Select one of the subcategories by clicking.
5. Select a command by clicking.
6. Click **Insert**. The command is inserted at current cursor position. Possible arguments are replaced by the user.
7. Repeat steps 3 to 6 to insert next command or click **Close** to exit the tool. The cursor position may also be moved while the **Insert SCIL Commands, Functions & Objects** dialog is open.

### 13.5.16 Syntax checking of a SCIL program

The syntax check, independent of the current cursor position, always starts at the beginning of the program. A successful syntax check displays **Syntax Checking Successful** in a message box. The syntax check command itself does not alter the code, changes are made by the user. The first encountered invalid code is displayed in a dialog as shown in the picture below. The erroneous line is shown in the dialog and it is also made the current line of the editor. The error is corrected in the text window and then checked by clicking the **Check** button in the **SCIL Syntax Error** dialog. If the correction passes the syntax check, the next error is displayed. This procedure continues until no more errors are found and the **Syntax Checking Successful** message is shown. To exit the dialog while errors still exist, click **Close**.

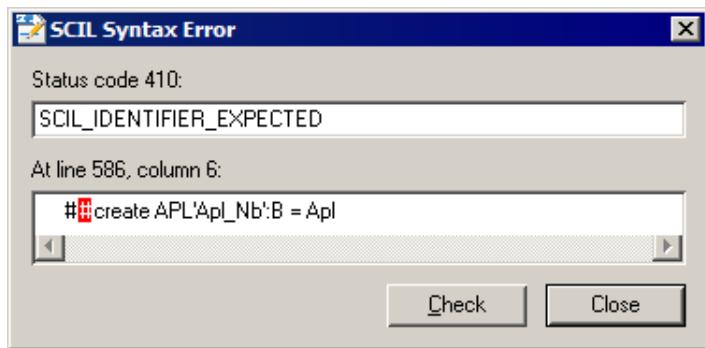


Figure 32: The Syntax check command shows a dialog like this when a syntax error is encountered

To check the syntax of a SCIL program:

1. Open a SCIL program.
2. Click **Syntax Check** on the SCIL menu. If an error is found a dialog showing the invalid code is displayed. If no errors are found in the program code, **Syntax Checking Successful** is displayed.
3. The erroneous line is shown in the **SCIL Syntax Error** dialog and the line is selected in the editor.
4. In the text window, correct the invalid code and click the **Check** button in the **SCIL Syntax Error** dialog. The next found error is displayed.
5. Repeat steps 3 and 4 until **Syntax Checking Successful** is displayed in a message box.



# Section 14 SCIL compiler

## 14.1 General

The SCIL programs of pictures and command procedures can be compiled for better performance. Compiling a SCIL program means that it is converted into an operating system independent format, which is then executed by a so called virtual SCIL machine. The compiled code is stored, in addition to the original SCIL code, in the picture or in the command procedure. Once a SCIL program is compiled, the compiled version is automatically used instead of the original SCIL code. The compilation is controlled by means of the corresponding tools, picture editor and command procedure tool.

## 14.2 Performance improvement

By compiling the SCIL code, the interpretation time is reduced to only a fraction of the original time. However, the total time needed to execute a SCIL statement depends very much on what the statement does, and thereby no single performance improvement between executing uncompiled and compiled SCIL code can be given. The performance improvement varies between no improvement and up to 50 times faster. Two extreme cases are shown below:

The following program is 50 times faster when compiled compared to uncompiled:

```
#loop_with i = 1 .. 1000
    #if TRUE #then #block
        #block_end
    #loop_end
```

The following program is not faster when compiled compared to uncompiled:

```
#pause 1
```

The picture change time of a typical single line diagram picture built with LIB500, which is compiled, is approximately 2/3 of the time of the uncompiled version.

## 14.3 Impact on SCIL programs

In most cases, a valid SCIL program executes exactly in the same way whether compiled or not (apart from the speed). However, there are some rare cases that must be considered:

1. A valid SCIL program may be impossible to compile.
2. A SCIL program may compile but generates a run-time error when run by the Virtual SCIL Machine.
3. The compiled and uncompiled program may generate different results.

Case 1 is the simplest one to handle as it is discovered during compilation. The program must be corrected and recompiled. Case 2 is more difficult, and careful retesting is required to point out the possible problems. Case 3 is the most difficult one to solve, but it is also an almost academic case which is hard to encounter unless deliberately written. Because of these possible incompatibilities, a compiled SCIL program must be thoroughly retested.

In most cases, the problems arise from wild usage of variable expansions. The cases are described in more detail below. Recommendations on how to avoid this kind of problems are given as well.

A compiled SCIL program may be harder to debug than an uncompiled one, because the source lines are not available at run-time. The error message field on the top left corner of a picture is not able to show the erroneous SCIL line (only the line number is given). The standard error dialog of VS objects is also unable to display the line that caused the error. It is recommended that a SCIL program is first debugged uncompiled, and after that compiled to a product version.

### 14.3.1 Programs that do not compile

There are two cases when a valid SCIL program does not compile:

- HMI commands do not compile. HMI commands are commands that are addressed to the HMI program started by !MODULENAME command (such as the old picture editor program PICG). In normal pictures they generate an error; 896 (PICO\_NO\_MODULE\_TO\_SEND\_MESSAGE). This restriction is a deliberate choice. HMI commands are seldom needed, and it is easy to write a HMI command by mistake, for example, by omitting character # or @ at the beginning of a line.
- Variable expansions are used in a way that hides the program structure from the reader.

Examples of the second case:

Example 1:

```
@CONDITION = "#IF A == B"  
'CONDITION' #THEN .DO_SOMETHING
```

Example 2:

```
@END = "_END"  
#BLOCK  
    .DO_SOMETHING  
#BLOCK'END'
```

Example 3:

```
@A = "1 + "  
@B = 'A' 2
```

### 14.3.2 Programs that generate run-time error

There are two cases when a program runs without errors when uncompiled but not when compiled:

- Alias checking is always performed by the Virtual SCIL Machine regardless of the revision compatibility switch (NO\_ALIAS\_CHECKING). The SCIL program design must be corrected to enable compilation if the compiled program fails with error code 580 (SCIL\_VARIABLE\_ALIASING\_ERROR).
- Variable expansions are used in a way that hides certain syntactical language elements from the reader.

Here are some examples of the second case:

Example 1:

```
@V = "AI (%INDEX)"  
@POWER = OBJECT:P'V'  
; Index expression hidden
```

This one works:

```
@V = "AI5"
@POWER = OBJECT:P'V'
```

**Example 2:**

```
@A = 1
@B = 2
@C = 'A'.'B' ;The compiler assumes that the right hand
;expression is a VS object or window
attribute reference
```

### 14.3.3 Programs that produce wrong results

If variable expansions are used in a way that hides the correct evaluation order of an expression, the results of compiled and uncompiled program may be different.

**Example:**

```
@A = "1 + 2"
@B = "3 + 4"
@C = 'A' * 'B'
```

As uncompiled, the meaning of the third line is

```
@C = 1 + 2 * 3 + 4
```

after the expansions, which is evaluated as  $1 + (2 * 3) + 4 = 11$ .

The compiler relies on what it sees as it generates byte code which expands and evaluates A first, then B and finally multiplies the two. The result is  $(1 + 2) * (3 + 4) = 21$ .

### 14.3.4 Recommendations

To avoid the potential problems described above (and to make SCIL programs more readable), following recommendations on the use of variable expansions may be given:

- Use direct variable access instead of expansion whenever possible.

Instead of 'A' + 'B', write %A + %B.

Instead of "'A'", write %A ( if A has a text value ).

Instead of "X'A'Z", write "X" + %A + "Z" ( if A has a text value ).

As another advantage, the direct variable access is faster than expansion even in uncompiled programs.

- Use variable expansion mainly for generating various identifiers.

Examples of good usage of expansions:

```
#SET 'LN':PBO1 = 1           ;As an object name

#SET ABC'POSTFIX':PBO1      ;As a part of an object name
= 1

@OLD_ERROR_STATE = ERROR_STATE

#ERROR IGNORE

.DO_SOMETHING

#ERROR 'OLD_ERROR_STATE'   ;As a command keyword value

.SET 'DIALOG'\BUTTON.TITLE = "Push me"      ;As a VS object name

!NEW_PIC                      ;As a picture name
'NEXT_PICTURE'
```

When variable expansions are used as recommended above, the compiled and uncompiled program always behave identically. Even most cases of other 'non-pathological' (or even 'pathological') use of expansions work as they are expected.

Examples:

```
@V = "PBO1"
#SET OBJECT:'V' = 1

@V = "OBJECT:PBO1"
#SET 'V' = 1

@V = "DIALOG\BUTTON"
.SET 'V'.TITLE = "Push me"
.SET 'V'_2.TITLE = "Push me too"

@NAME = "ABC"
@'NAME' = %'NAME' + 1
```

# Appendix A ODBC ERROR CODES

## 1.1 About this appendix

This appendix lists the ODBC error codes which the SCIL SQL interface may return.

## 1.2 General

The character string value consists of a two character class value followed by a three character subclass value. A class value of "01" indicates a warning and is accompanied by a return code of SQL\_SUCCESS\_WITH\_INFO. The class values other than "01", except for the class "IM", indicate an error and are accompanied by a return code of SQL\_ERROR. The class "IM" is specific to warnings and errors that derive from the implementation of ODBC itself. The subclass value "000" in any class is for implementation defined conditions within the given class.

Note 1 Although successful execution of a function is normally indicated by a return value of SQL\_SUCCESS, the SQLSTATE 00000 also indicates success.

Note 2 Although the SCILSQL interface has only eight functions the user can call, it includes several different ODBC functions. ODBC error codes may also refer to those hidden functions.

## 1.3 List of codes

SQLSTATE	Error
01000	General warning
01002	Disconnect error
01004	Data truncated
01006	Priviledge not revoked
01S02	Option value changed
01S03	No rows updated or deleted
01S04	More than one row updated or deleted
07001	Wrong number of parameters
07006	Restricted data type attribute violation
08001	Unable to connect to data source
08002	Connection in use
08003	Connection not open
08004	Data source rejected establishment of connection
08007	Connection failure during transaction
08S01	Communication link failure
21S01	Insert value list does not match column list
21S02	Degree of derived table does not match column list
22003	Numeric value out of range
22005	Error in assignment
22008	Datetime field overflow

Table continues on next page

SQLSTATE	Error
22012	Division by zero
23000	Integrity constraint violation
24000	Invalid cursor state
25000	Invalid transaction state
28000	Invalid authorization specification
34000	Invalid cursor name
37000	Syntax error or access violation
40001	Serialization failure
42000	Syntax error or access violation
IM001	Driver does not support this function
IM002	Data source name not found and no default driver specified
IM003	Specified driver could not be loaded
IM004	Driver's SQLAllocEnv failed
IM005	Driver's SQLAllocConnect failed
IM006	Driver's SQLSetConnect-Option failed
IM009	Unable to load translation DLL
IM013	Trace file error
S0001	Base table or view already exists
S0002	Base table not found
S0011	Index already exists
S0012	Index not found
S0021	Column already exists
S0022	Column not found
S1000	General error
S1001	Memory allocation failure
S1002	Invalid column number
S1003	Program type out of range
S1008	Operation cancelled
S1009	Invalid argument value
S1010	Function sequence error
S1011	Operation invalid at this time
S1012	Invalid transaction operation code specified
S1090	Invalid string or buffer length
S1092	Option type out of range
S1109	Invalid cursor position
S1C00	Driver not capable
S1T00	Timeout expired

# Appendix B    PARAMETER FILES

## 1.1    About this appendix

This appendix describes the parameter files used in SYS600. First, it gives a general description, then a detailed description in BNF format.

## 1.2    General

Parameter files contain data in a Windows ini-file style format. A parameter file consists of any number of sections and any number of comment lines. Each section consists of a section header and any number of keys: value pairs, empty lines and comment lines. However, the total number of lines in a file is limited to the maximum length of SCIL vector.

Example:

```
[SECTION1]
KEY=VALUE
;this is a comment
[SECTION2]
KEY=VALUE
```

A section begins with a header, the name of the section enclosed in brackets, e.g. [section name], and ends at the beginning of another section or at the end of the file. The enclosing brackets ([]) are required, and the left bracket must be in the leftmost column.

Comment lines begin with a semicolon (;).

## 1.3    BNF description

In the following description:

- NL represents new line characters (ASCII CR and LF)
- SP represents any number of space characters
- CH represents any non-control character (ASCII codes 32 to 255)
- NC represents any visible character, except ; [ ] and =

```
ini_file ::= {comment_line}* {section}*
section ::= section_header {item}*
section_header ::= [section_name] NL
section_name ::= name
item ::= NL | key_line | comment_line
comment_line ::= SP ; character_string NL
key_line ::= SP key_name SP = SP key_value NL
```

```
key_value ::= character_string
character_string ::= {CH}*
key_name ::= name
name ::= NC{extended_name_char}*
extended_name_char ::= SP NC
```

# Index

- ATTRIBUTE_NAMES.....	70	BCD_TO_INTEGER.....	144, 184
- CHILD_OBJECTS.....	70	BG.....	329
- COMPILED.....	70	BIN.....	144, 185
- FILE_REVISION.....	71	BIN_SCAN.....	144, 185
- FLAG_FOR_EXECUTION.....	69	BINARY_SEARCH.....	144, 197
- OBJECT_CLASS.....	71	Binary Coded Decimal numbers.....	184, 188
- OBJECT_NAME.....	71	Binary files.....	74
- OBJECT_PATH.....	71	BIT.....	144, 194
- QUEUE_FOR_EXECUTION.....	69	BIT_AND.....	144, 194
- SG_GEOMETRY.....	71	BIT_CLEAR.....	144, 194
- SOURCE_FILE_NAME.....	72	BIT_COMPL.....	144, 195
- VARIABLE_NAMES.....	72	BIT_MASK.....	144, 195
<b>A</b>		BIT_OR.....	144, 195
ABS.....	143, 161	BIT_SCAN.....	144, 185
Absolute value.....	143, 161	BIT_SET.....	144, 196
Accuracy.....	42	BIT_STRING.....	144, 196
ADD_INTERLOCKED.....	143, 318	BIT_XOR.....	144, 196
Addition.....	86, 87	Bit function.....	194
AEP_PROGRAMS.....	143, 213	Bit string.....	41, 44, 194
Alarm buffer.....	55	BLOCK.....	98, 99
Alarm list.....	55	BLOCK_END.....	98, 99
Alarm picture.....	97, 128	Boolean.....	43, 89
Alarm picture queue.....	128	BOX.....	322, 323
AM.....	329	Byte string.....	45
AND.....	91	<b>C</b>	
APL.....	51, 52	Canceling in SCIL Program Editor.....	359
APL_OPCNAM.SDB.....	297	Canvas.....	321, 322, 325, 326, 336, 337, 341
APPEND.....	143, 197	CANVAS CURRENT.....	337
Application.....	51, 54	CANVAS PARENT.....	337
APPLICATION_ALARM_COUNT.....	143, 238	CANVAS ROOT.....	337
APPLICATION_ALARM_LIST.....	55, 143, 239	CAP_STYLE.....	329
APPLICATION_OBJECT_ATTRIBUTES.....	143, 230	CAPITALIZE.....	144, 185
APPLICATION_OBJECT_COUNT.....	143, 231	CASE_END.....	98, 99
APPLICATION_OBJECT_EXISTS.....	143, 232	CASE command.....	98, 99
APPLICATION_OBJECT_LIST.....	143, 232	CASE function.....	144, 156
APPLICATION_OBJECT_SELECT.....	143, 235	Characters.....	38
Application engineering.....	25	Child object.....	64
Application objects.....	49	CHOOSE.....	144, 156
ARC.....	322, 329	CIRCLE.....	322, 323
ARC_MODE.....	329	CLASSIFY.....	144, 198
ARCCOS.....	143, 162	CLOCK.....	144, 169
ARCSIN.....	143, 162	CLOSE.....	126, 354
ARCTAN.....	143, 162	CLOSE_FILE.....	118
Argument.....	85, 98, 143, 209	COLLECT.....	144, 186
ARGUMENT_COUNT.....	143, 209	Color.....	331, 333
Argument list.....	142	COLOR_IN.....	335, 336
Arguments.....	95, 143, 209	Command.....	29, 85
Arithmetical operator.....	86	Command procedure.....	33, 54, 59
ASCII.....	143, 183	Comma Separated Value.....	284
ASCII_CODE.....	143, 184	Commenting in SCIL Program Editor.....	355
Assignment.....	79	Comment Mark.....	355
Assignment statement.....	98	Comments.....	37
Attribute.....	46, 50, 51, 54	Communication system object.....	50, 52
ATTRIBUTE_EXISTS.....	143, 207	Communication unit.....	50, 52
AUDIO_ALARM.....	143, 319	COMPILE.....	144, 218
<b>B</b>		Component.....	321, 327, 328, 333, 334, 335, 336
BACKGROUND.....	329	CONSOLE_OUTPUT.....	144, 213
Background color.....	329	CONTINUE.....	101
Background program.....	32	COORDINATE_SYSTEM.....	337
BASE_SYSTEM_OBJECT_LIST.....	143, 236	Coordinate system.....	321
Base system object.....	50, 51	Copying in SCIL Program Editor.....	355
BCD.....	184, 188	COS.....	144, 162
<b>C</b>		CREATE.....	108, 123, 353

CREATE\_FILE..... 119  
 CS..... 329  
 CSR\_BOL..... 136  
 CSR\_EOL..... 136  
 CSR\_LEFT..... 136  
 CSR\_RIGHT..... 136  
 CSV\_TO\_SCIL..... 144, 285  
 CSV functions..... 284  
 CUMULATE..... 144, 198  
 CURRENT..... 327, 328  
 CURSOR\_POS..... 83

**D**

DASH\_LIST..... 329  
 DASH\_OFFSET..... 330  
 Dashed line..... 329  
 DATA\_FETCH..... 144, 241  
 DATA\_MANAGER..... 144, 260  
 DATA\_STORE..... 145, 242  
 DATA\_TYPE..... 145, 157  
 Data object..... 34, 54, 58  
 Data type..... 41  
 DATE..... 145, 170  
 DAY..... 145, 170  
 Day of Week..... 146, 170  
 Day of Year..... 146, 170  
 DDE\_CONNECT..... 145, 287  
 DDE\_DISCONNECT..... 145, 288  
 DDE\_EXECUTE..... 145, 290  
 DDE\_POKE..... 145, 289  
 DDE\_REAL..... 145, 291  
 DDE\_REQUEST..... 145, 289  
 DDE\_VECTOR..... 145, 291  
 DDE client..... 286  
 DDE protocol..... 36, 290  
 DDE Server..... 291  
 Debugging..... 229  
 DEC..... 145, 186  
 DEC\_SCAN..... 145, 186  
 Default path names..... 117  
 DELETE..... 108, 123, 124  
 DELETE\_ATTRIBUTE..... 145, 207  
 DELETE\_FILE..... 119  
 DELETE\_PARAMETER..... 145, 263  
 Deleting in SCIL Program Editor..... 355  
 Dialogs..... 25  
 Dialog systems..... 64  
 DIRECTORY\_MANAGER COPY..... 145, 276  
 DIRECTORY\_MANAGER COPY CONTENTS.....  
145, 276  
 DIRECTORY\_MANAGER CREATE..... 145, 275  
 DIRECTORY\_MANAGER DELETE..... 146, 275  
 DIRECTORY\_MANAGER DELETE\_CONTENTS...  
146, 275  
 DIRECTORY\_MANAGER EXISTS..... 146, 276  
 DIRECTORY\_MANAGER GET\_ATTRIBUTES.....  
146, 277  
 DIRECTORY\_MANAGER LIST..... 146, 275  
 DIRECTORY\_MANAGER MOVE..... 146, 277  
 DIRECTORY\_MANAGER RENAME..... 146, 277  
 Directory name..... 73  
 Directory tag..... 73  
 DIV..... 86, 87  
 Division..... 86, 87  
 DL..... 329  
 DO..... 98, 100, 330  
 DO function..... 146, 209  
 DOW..... 146, 170  
 DOY..... 146, 170  
 Draw program..... 32

DRIVE\_MANAGER EXISTS..... 146, 274  
 DRIVE\_MANAGER GET\_ATTRIBUTES..... 146, 274  
 DRIVE\_MANAGER GET\_DEFAULT..... 146, 274  
 DRIVE\_MANAGER LIST..... 146, 274  
 DUMP..... 146, 157  
 duration curves..... 198, 201

**E**

EDIT..... 146, 187  
 Element..... 45  
 ELEMENT\_LENGTH..... 146, 157  
 ELLIPSE..... 322, 323  
 ELSE..... 98, 102  
 ELSE\_IF..... 98, 102  
 Endpoints..... 329  
 ENTER..... 136  
 ENTER\_POS..... 83  
 ENVIRONMENT..... 146, 214  
 EQUAL..... 146, 158  
 Equal to..... 89  
 ERASE..... 130  
 ERROR\_STATE..... 146, 210  
 ERROR\_CONTINUE..... 98, 101  
 ERROR\_EVENT..... 98, 101  
 Error handling policy..... 96, 101  
 ERROR\_IGNORE..... 98, 101  
 ERROR\_RAISE..... 98, 101  
 ERROR\_STOP..... 98, 101  
 EVALUATE..... 146, 158  
 EVEN..... 146, 162  
 EVENT..... 101  
 Event channel..... 54, 61  
 Event handling object..... 57  
 Event list..... 55  
 Event object..... 54, 62  
 EXEC..... 59, 109  
 EXEC\_AFTER..... 109  
 Execute..... 287  
 Execution mode..... 28  
 Execution time..... 60  
 Exit program..... 32  
 EXP..... 146, 162  
 Exponent..... 87  
 Exponential operator..... 86  
 Export..... 359  
 Expression..... 29, 82, 98, 133

**F**

FALSE..... 43  
 FAST\_PIC..... 126, 127  
 FG..... 331  
 Fictitious process objects..... 55  
 FILE\_LOCK\_MANAGER..... 146, 263  
 FILE\_MANAGER COPY..... 146, 278  
 FILE\_MANAGER DELETE..... 146, 278  
 FILE\_MANAGER EXISTS..... 146, 278  
 FILE\_MANAGER GET\_ATTRIBUTES..... 146, 279  
 FILE\_MANAGER LIST..... 147, 278  
 FILE\_MANAGER MOVE..... 147, 279  
 FILE\_MANAGER RENAME..... 147, 279  
 File name..... 73  
 File naming..... 72  
 File tag..... 73  
 FIND\_ELEMENT..... 147, 199  
 Finding..... 356  
 FLUSH..... 341  
 FM\_APPLICATION\_DIRECTORY..... 280  
 FM\_APPLICATION\_FILE..... 280  
 FM\_COMBINE..... 147, 280  
 FM\_COMBINE\_NAME..... 147, 280

FM_DIRECTORY.....	147, 281
FM_DRIVE.....	147, 281
FM_EXTRACT.....	147, 281
FM_FILE.....	147, 282
FM_REPRESENT.....	147, 282
FM_SCIL_DIRECTORY.....	147, 282
FM_SCIL_FILE.....	147, 283
FM_SCIL_REPRESENT.....	147, 283
FM_SPLIT_NAME.....	147, 283
FONT.....	331, 333, 334, 335, 336
Font number.....	334
FOREGROUND.....	323, 325, 331, 332, 333
Format picture.....	114
Free type object.....	54
FT.....	331
FU.....	331
Function.....	29, 85, 142, 331, 333
Function call.....	142
Function key program.....	32
<b>G</b>	
GC.....	328, 335
General SCIL commands.....	95
GET.....	110
GET_STATUS.....	147, 159
Global variables.....	79
Go To.....	358
Graphical element.....	321, 322, 328, 336
Graphics command.....	321, 327, 336
Graphics commands.....	95
Graphics context...321, 322, 326, 327, 328, 333, 334, 335, 341	
Greater than.....	89
Greater than or equal to.....	89
Group.....	56
<b>H</b>	
HEADER.....	136
HELP.....	136
HEX.....	147, 187
HEX_SCAN.....	147, 187
HIGH.....	147, 149, 199
HIGH_INDEX.....	147, 149, 200
HIGH_PRECISION_ADD.....	147, 163
HIGH_PRECISION_DIV.....	147, 163
HIGH_PRECISION_MUL.....	147, 163
HIGH_PRECISION_SHOW.....	147, 164
HIGH_PRECISION_SUB.....	147, 164
HIGH_PRECISION_SUM.....	148, 164
High precision arithmetics.....	163
HISTORY_DATABASE_MANAGER.....	55
HISTORY_DATABASE_MANAGER CLOSE...148, 245	
HISTORY_DATABASE_MANAGER GET_PARAM ETERS....	148, 249
HISTORY_DATABASE_MANAGER OPEN....	148, 245
HISTORY_DATABASE_MANAGER QUERY...148, 249	
HISTORY_DATABASE_MANAGER READ....	148, 250
HISTORY_DATABASE_MANAGER SET_ATTRIB UTES....	148, 248
HISTORY_DATABASE_MANAGER SET_COMM ENT....	148, 250
HISTORY_DATABASE_MANAGER SET_CONDI TION....	148, 248
HISTORY_DATABASE_MANAGER SET_DIRECT ION....	148, 247
HISTORY_DATABASE_MANAGER_SET_DIRECT ORY....	148, 246
HISTORY_DATABASE_MANAGER_SET_ORDER ....	148, 247
HISTORY_DATABASE_MANAGER_SET_PERIOD ....	148, 245
HISTORY_DATABASE_MANAGER_SET_TIMEO UT....	148, 248
HISTORY_DATABASE_MANAGER_SET_WINDO W....	148, 246
HISTORY_DATABASE_MANAGER_WRITE....	148, 251
History database.....	55
HOD.....	148, 170
HOUR.....	148, 171
HOY.....	148, 171
HR_CLOCK.....	148, 171
<b>I</b>	
Identifiers.....	39
IF command.....	98, 102
IF function.....	148, 159
IGNORE.....	101
IMAGE.....	322, 324
Import.....	359
Index.....	51, 54, 56
INIT_QUERY.....	110, 244
Initialization time.....	60
INPUT_KEY.....	136
INPUT_POS.....	137
INPUT_VAR.....	137
INSERT_ELEMENT.....	148, 200
Inserting SCIL commands, functions and obj ects....	359
INT_PIC.....	126, 128
Integer.....	41
INTEGER_TO_BCD.....	148, 188
INTERP.....	148, 201
INVERSE.....	148, 201
IP_PROGRAMS.....	148, 214
Item.....	36
Item name.....	287
<b>J</b>	
JOIN.....	148, 189
JOIN_STYLE.....	329, 332
JS.....	332
<b>K</b>	
KEY_POS.....	83
KEYED_FILE_MANAGER.....	148, 264
Keyed files.....	74
<b>L</b>	
LAST_PIC.....	126, 128
Less than.....	89
Less than or equal to.....	89
Level parameter.....	98, 130, 133
LIB500.....	25
Library representation.....	98, 135
LIN.....	52
LINE.....	322, 324, 329, 332
LINE_STYLE.....	332
LINE_WIDTH.....	332
List.....	41, 46, 114, 149, 208
LIST_ATTR.....	149, 208
List aggregate.....	46
LN.....	149, 165
LOAD.....	123, 124
LOCAL.....	98, 102
LOCAL_TIME.....	149, 171

LOCAL\_TIME\_ADD..... 149, 171  
 LOCAL\_TIME\_INFORMATION..... 149, 172  
 LOCAL\_TIME\_INTERVAL..... 149, 172  
 LOCAL\_TO\_SYS\_TIME..... 149, 173  
 LOCAL\_TO\_UTCTIME..... 149, 173  
 Local variables..... 79  
 LOCATE..... 149, 189  
 Logging profile..... 54  
 LOGICAL\_MAPPING..... 149, 255  
 Logical names..... 39  
 Logical operator..... 86, 91  
 Logical path..... 116  
 Logical representation library..... 97, 117  
 LOOP..... 98, 103  
 LOOP\_END..... 98, 103, 105  
 LOOP\_EXIT..... 98, 105  
 LOOP\_WITH..... 98, 105  
 LOW..... 147, 149, 199  
 LOW\_INDEX..... 147, 149, 200  
 LOWER\_CASE..... 149, 190  
 LS..... 332  
 LW..... 332

**M**

Main picture..... 31  
 MAX..... 149, 165  
 MAX\_APPLICATION\_NUMBER..... 149, 218  
 MAX\_BIT\_STRING\_LENGTH..... 149, 218  
 MAX\_BYTE\_STRING\_LENGTH..... 149, 218  
 MAX\_INTEGER..... 149, 218  
 MAX\_LINK\_NUMBER..... 149, 218  
 MAX\_LIST\_ATTRIBUTE\_COUNT..... 149, 219  
 MAX\_MONITOR\_NUMBER..... 149, 219  
 MAX\_NODE\_NUMBER..... 149, 219  
 MAX\_OBJECT\_NAME\_LENGTH..... 149, 219  
 MAX\_PICTURE\_NAME\_LENGTH..... 149, 219  
 MAX\_PRINTER\_NUMBER..... 149, 219  
 MAX\_PROCESS\_OBJECT\_INDEX..... 149, 219  
 MAX REPRESENTATION\_NAME\_LENGTH. 149,  
                                       219  
 MAX\_STATION\_NUMBER..... 149, 220  
 MAX\_STATION\_TYPE\_NUMBER..... 149, 220  
 MAX\_TEXT\_LENGTH..... 149, 220  
 MAX\_VECTOR\_LENGTH..... 149, 220  
 MAX\_WINDOW\_NAME\_LENGTH..... 149, 220  
 MEAN..... 150, 202  
 MEMORY\_POOL\_USAGE..... 150, 214  
 MEMORY\_USAGE..... 150, 210  
 MERGE\_ATTRIBUTES..... 150, 208  
 Method..... 33, 50  
 Method call..... 125  
 Method calls..... 66  
 MIN..... 150, 165  
 MIN\_INTEGER..... 150, 220  
 MINUTE..... 150, 173  
 Mix..... 333, 334  
 MOD..... 86, 87  
 MODIFY..... 111, 125  
 MON..... 51  
 Monitor..... 97, 126, 327, 328  
 MONTH..... 150, 173  
 MOUSE..... 339  
 MOUSE\_DISCARD..... 340  
 Mouse input..... 339  
 MOUSE OFF..... 339  
 MOUSE ON..... 339  
 Moving in SCIL Program Editor..... 355  
 Multiplication..... 86, 87

**N**

NA..... 333  
 Name..... 51, 54, 333  
 NAME\_HIERARCHY..... 251  
 Named program..... 32, 68, 69  
 Natural logarithm..... 149, 165  
 NET..... 52  
 NETWORK\_TOPOLOGY\_MANAGER.... 150, 256  
 Network Topology..... 256  
 NEW\_PIC..... 126, 128  
 New Folder..... 354, 359  
 NOD..... 52  
 Normal Mode..... 28  
 NOT..... 91  
 Notification window..... 144, 213

**O**

Object..... 29, 49, 53, 96, 108  
 OBJECT\_ATTRIBUTE\_INFO..... 150, 220  
 Object notation..... 51, 54  
 Object query..... 96, 110  
 OCT..... 150, 190  
 OCT\_SCAN..... 150, 190  
 Octal number..... 41  
 ODBC functions..... 292  
 ODD..... 150, 166  
 ON..... 62, 98, 105  
 ON ERROR..... 98, 106  
 ON KEY\_ERROR..... 98, 106  
 OPC\_AE\_ACKNOWLEDGE..... 150, 299  
 OPC\_AE\_NAMESPACE..... 151, 300  
 OPC\_AE\_REFRESH..... 151, 301  
 OPC\_AE\_SERVERS..... 151, 301  
 OPC\_AE\_VALIDATE..... 151, 302  
 OPC\_DA\_NAMESPACE..... 151, 303  
 OPC\_DA\_REFRESH..... 151, 305  
 OPC\_DA\_SERVERS..... 151, 305  
 OPC\_NAME\_MANAGER..... 151, 298  
 OPC Name Database..... 297  
 OPEN\_FILE..... 119  
 Operands..... 85  
 Operator..... 85, 86, 91  
 OPS\_CALL..... 151, 211  
 OPS\_NAME..... 151, 215  
 OPS\_PROCESS..... 151, 211  
 OR..... 91  
 OTHERWISE..... 98, 99

**P**

PACK\_STR..... 151, 190  
 PACK\_TIME..... 151, 173  
 PAD..... 151, 191  
 Parameter files..... 73  
 PARENT..... 327, 328  
 Parent object..... 64  
 PARSE\_FILE\_NAME..... 151, 265  
 Part picture..... 31  
 PATH..... 116, 151, 266  
 PATHS..... 151, 266  
 PAUSE..... 98, 107  
 Pending..... 341  
 PEND OFF..... 341  
 PEND ON..... 341  
 Peripherals equipment..... 52  
 PHYSICAL\_MAPPING..... 151, 256  
 PIC\_NAME..... 83  
 PICK..... 151, 202  
 Picture..... 25, 67, 98, 134  
 Picture commands..... 95  
 Picture Editor..... 31

Picture handling commands.....	126
Picture path.....	67
Picture programs.....	32
Picture queue.....	126, 128
POINT.....	322, 325
Poke.....	287
Polygon.....	325
POLYLINE.....	322, 325
POP.....	341
Predefined picture variable.....	82
PRI.....	51, 52
PRINT.....	114, 115
PRINT_TRANSPARENT.....	151, 309
PRINT_SET.....	152, 312
Printout.....	96, 114, 115
Printout function.....	151, 309
Process database.....	54
Process object.....	53, 55
Process query.....	96, 110
PROGRAM.....	136
PUSH.....	341
<b>R</b>	
RANDOM.....	152, 166
random number.....	152, 166
random order.....	153, 205
READ.....	120
READ_BYTES.....	152, 266
READ_COLUMNS.....	152, 267
READ_KEYS.....	120
READ_NEXT.....	121
READ_PARAMETER.....	152, 267
READ_PREV.....	121
READ_TEXT.....	152, 268
Read-only.....	353
Read-only Mode.....	28
Real.....	41, 42
RECALL_PIC.....	126, 129
Redoing in SCIL Editor.....	359
REGISTRY.....	152, 216
Relational operator.....	86, 89, 90
Remote terminal unit.....	53
REMOVE.....	121
REMOVE_DUPLICATES.....	152, 203
RENAME_FILE.....	122
REP_LIB.....	116, 117, 152, 268
REP_LIBS.....	152, 269
REPLACE.....	152, 192, 357
Replacing in SCIL Program Editor.....	356
Report database.....	54
Reporting object.....	54
Request.....	287
RESET.....	138, 139
RESTORE.....	126, 129
RETURN.....	98, 107
REVERSE.....	152, 203
REVISION_COMPATIBILITY.....	152, 212
ROOT.....	327, 328
ROUND.....	152, 167
RTU.....	53, 306
RTU_ADDR.....	152, 306
RTU_AREAL.....	152, 307
RTU_ATIME.....	152, 307
RTU_BIN.....	152, 307
RTU_HEXASC.....	152, 307
RTU_INT.....	152, 308
RTU_KEY.....	152, 308
RTU_MSEC.....	152, 308
RTU_OA.....	152, 308
RTU_REAL.....	152, 308
RTU_TIME.....	152, 309
RTU_FUNCTION.....	306
RUBOUT.....	138
RUBOUT_BOL.....	138
RUBOUT_CUR.....	138
RUBOUT_EOL.....	138
<b>S</b>	
Save.....	354
Scale.....	53, 57, 152, 319
SCALING.....	338
Scaling factor.....	321, 336, 338, 339, 341
SCIL.....	25
SCIL_HOST.....	152, 216
SCIL_LINE_NUMBER.....	153, 229
SCIL_TO_CSV.....	153, 286
SCIL coordinate.....	337
SCIL database.....	76
SCIL Data Derivation Language.....	91
SCIL defined printout.....	309
SCIL Editor.....	349
SCIL program.....	37
SCIL Program Editor.....	31, 355
Scope.....	328
SDDL.....	91
SEARCH.....	112, 238
Searching.....	356
SECOND.....	153, 174
SELECT.....	153, 203
Selecting Text.....	355
SEND_PIC.....	138
SEPARATE.....	153, 192
Service.....	36
Service name.....	287
Session.....	312
SET.....	55, 113, 125, 126
SET_CLOCK.....	153, 174
SET_EVENT_LIST_USER_NAME.....	153, 312
SET_LANGUAGE.....	153, 228
SET_LOCAL_TIME.....	153, 174
SET_RANDOM_SEED.....	153, 167
SET_STATUS.....	153, 160
SET_SYS_TIME.....	153, 174
SET_TIME.....	98, 107
SET_UTCTIME.....	153, 175
SHADOW_FILE.....	153, 269
SHOW.....	130, 131
SHOW_BACK.....	130, 131
SHUFFLE.....	153, 205
SIN.....	153, 167
Sine.....	162
Snapshot variables.....	80
SORT.....	153, 205
SPACOM.....	153, 283, 284
SPREAD.....	153, 205
SQL.....	292
SQL_BEGIN_TRANSACTION.....	153, 295
SQL_COMMIT.....	153, 296
SQL_CONNECT.....	153, 292
SQL_DISCONNECT.....	153, 293
SQL_EXECUTE.....	153, 293
SQL_FETCH.....	153, 294
SQL_FREE_STATEMENT.....	153, 295
SQL_ROLLBACK.....	153, 296
SQL statement.....	294
SQRT.....	153, 167
Square root.....	153, 167
STA.....	51, 52
Stacking order.....	130
Start program.....	32

Statement.....	27, 29	UPDATE.....	126, 129
Station.....	53	Update program.....	32
STATUS.....	153, 213	Update time interval.....	97, 129
STATUS_CODE.....	153, 222	UPPER_CASE.....	154, 193
STATUS_CODE_NAME.....	154, 223	USER.....	327, 328
Status code.....	41, 58, 102, 147, 153, 159, 160	User Interface Objects.....	49, 67
STOP.....	101	USM_ADDRESS.....	154, 313
STORE_PIC.....	126, 129	USM_AOR_DATA.....	154, 313
String function.....	183	USM_AUTHORIZATION_LEVEL.....	155, 313
STY.....	52	USM_AUTHORIZATION_LEVEL_FOR_OBJECT.....	155, 313
Sub-picture.....	31	USM_AUTHORIZATIONS.....	155, 314
SUBSTR.....	154, 192	USM_CHANGE_PASSWORD.....	155, 314
Subtraction.....	86, 87	USM_IS_NEW_APPLICATION.....	155, 314
SUM.....	154, 206	USM_LOGIN.....	155, 315
SUM_NEG.....	154, 206	USM_LOGOUT.....	155, 315
SUM_POS.....	154, 206	USM_PASSWORD_CHANGE.....	155, 315
Syntax check.....	360	USM_PASSWORD_POLICY.....	155, 316
SYS.....	51	USM_SELECT_ROLE.....	155, 316
SYS_TIME.....	154, 175	USM_SESSION_ATTRIBUTES.....	155, 317
SYS_TIME_ADD.....	154, 175	USM_SESSION_ID.....	155, 317
SYS_TIME_INTERVAL.....	154, 176	USM_SESSIONS.....	155, 317, 318
SYS_TIME.PAR.....	179	USM_USER_LANGUAGE.....	155, 318
SYS_TO_LOCAL_TIME.....	154, 176	USM_USER_NAME.....	155, 318
SYS_TO_UTCTIME.....	154, 176	USM_USER_ROLE.....	155, 318
System Objects.....	49, 50	USM_USER_ROLES.....	155, 318
<b>T</b>		UTC_TIME.....	155, 181
tangent.....	162	UTC_TIME_ADD.....	155, 181
Text.....	41, 44, 322, 326	UTC_TIME_INTERVAL.....	155, 182
TEXT_READ.....	154, 269	UTC_TO_LOCAL_TIME.....	155, 182
Text files.....	73	UTC_TO_SYS_TIME.....	155, 182
Text Selecting.....	355		
THEN.....	98, 102	<b>V</b>	
Time.....	41, 154, 177	VALIDATE.....	155, 223
TIME_SCAN.....	154, 177	VALIDATE_OBJECT_ADDRESS.....	155, 223
TIME_ZONE_RULES.....	154, 178	Variable.....	29, 79
Time channel.....	35, 54, 60	VARIABLE_NAMES.....	155, 213
time functions.....	168	Variable Assignment.....	81
TIMEMS.....	154, 180	Variable expansion.....	82
Time of day.....	154, 181	Variable object.....	54, 63
TIMEOUT.....	154, 284	Vector.....	41, 45, 155, 207
Time-out.....	284	Vector aggregate.....	45
TIMES.....	154, 180	Vector function.....	197
Time stamp.....	58	VIDEO_NR.....	83
TOD.....	154, 181	Visual SCIL.....	25
TODMS.....	154, 181	Visual SCIL Commands.....	95
TODS.....	154, 181	Visual SCIL objects.....	64, 123
TOGGLE_MOD.....	138		
Topic.....	36	<b>W</b>	
Topic name.....	287	WEEK.....	155, 182
TRACE_BEGIN.....	154, 230	WHEN.....	98, 99
TRACE_END.....	154, 230	WIN_BG_COLOR.....	130, 132
TRACE_PAUSE.....	154, 230	WIN_CREATE.....	130, 132
TRACE_RESUME.....	154, 230	WIN_INPUT.....	130, 133
Tracing.....	339	WIN_NAME.....	130, 134
TRANSLATE.....	154, 228	WIN_PIC.....	130, 134
TRANSLATION.....	154, 229	WIN_POS.....	130, 135
Transparent printout.....	309	WIN REP.....	130, 135
TREND.....	154, 207	Window.....	97, 130
TRUE.....	43	Window level.....	130
TRUNC.....	154, 167	Workstation.....	52
Type.....	51, 54, 354	WRITE.....	122
TYPE_CAST.....	154, 161	WRITE_BYTES.....	155, 270
<b>U</b>		WRITE_COLUMNS.....	155, 270
Uncommenting.....	355	WRITE_PARAMETER.....	155, 271
Undo.....	354	WRITE_TEXT.....	155, 271
Undoing in SCIL Program Editor.....	359		
Unequal.....	89	<b>X</b>	
UNPACK_STR.....	154, 193	XOR.....	91

**Y**  
YEAR.....155, 183





---

**Hitachi ABB Power Grids**  
**Grid Automation Products**  
PL 688  
65101 Vaasa, Finland



Scan this QR code to visit our website

<https://hitachiabb-powergrids.com/microscadax>