



version 6.2
Development System

SCL LANGUAGE REFERENCE

UPMACS Communications Inc.

Table of Contents

TABLE OF CONTENTS.....	i
THE SCL PROGRAMMING LANGUAGE	12
INTRODUCTION	12
LANGUAGE COMPONENTS	13
REMARKS	13
LITERAL VALUES.....	13
Numbers	13
Strings.....	13
CONSTANTS.....	14
RESERVED VARIABLES	14
USER VARIABLES	14
ARRAYS	15
Indices	15
Creating Arrays	15
Accessing Elements.....	15
FUNCTIONS.....	16
MATHEMATICAL EXPRESSIONS	16
Operator Priority.....	17
ASSIGNMENTS.....	18
COMMANDS	18
PROGRAMMING METHODS	19
PROGRAM ARGUMENTS	19
CONDITIONAL STATEMENTS	19
Single Command Conditions.....	19
IF-THEN-ENDIF Blocks.....	19
LINE NUMBERS, JUMPS, AND SUBROUTINES	21
Line Numbers	21
Jumps.....	21
Subroutines.....	22
Multibranching	22
LOOPS	22
WHILE-DO Loops.....	22
REPEAT-UNTIL Loops	23
FOR-NEXT Loops.....	24
MESSAGES TO THE USER.....	25
When to Use Which Command.....	26
DIALOGS.....	28
How Dialogs Work.....	28
Dialog Items	28

Dialog Buttons.....	29
The Result Variable.....	29
The Button Callback.....	30
▪ Perform validation of data the user entered.....	30
▪ Perform an action that does not close the dialog.....	30
TIME AND DATE.....	31
FILE INPUT AND OUTPUT AND NETWORK CONNECTIONS.....	31
File Input and Output.....	31
Network Connections.....	32
DECODING AND ENCODING DATA.....	32
Decoding or Encoding a Single Value.....	32
Decoding a Sequence of Values from a String.....	33
Encoding a Sequence of Values to a String.....	33
SERIAL COMMUNICATION.....	34
Sending Commands.....	34
Synchronizing Port Access.....	35
PROGRAMS FOR SOURCES, CHECKSUMS, AND SABUS RESPONSE DATA.....	35
Accessing the Data (Processor Sources and Checksums Only).....	35
Specifying the Data Object/Register Value or Checksum.....	36
▪ Checksums.....	36
▪ Serial Data Objects and Registers.....	36
▪ SABus Response Data Objects.....	37
Restrictions on Functions and Commands.....	37
PROGRAMS FOR SABUS COMMANDS.....	37
Restrictions on Functions and Commands.....	38
DEVICE DRIVER PROGRAMS.....	38
▪ Registers and legacy parameters.....	38
▪ Serial ports and devices.....	39
▪ SCL programs.....	39
INVOKING SCL PROGRAMS FROM WITHIN AN SCL PROGRAM.....	39
Executing Programs On a Remote Computer.....	39
RTS CONTROLS.....	40
Restrictions on Functions and Commands.....	40

RESERVED VARIABLE REFERENCE

41

SERIAL COMMUNICATION RESERVED VARIABLES.....	41
▪ THE DRVSUCCESS% RESERVED VARIABLE.....	41
▪ THE DRVTIMEOUT% RESERVED VARIABLE.....	41
▪ THE DRVEERROR% RESERVED VARIABLE.....	42
▪ THE DRVDATA\$ RESERVED VARIABLE.....	42
▪ THE DRVEERROR RESERVED VARIABLE.....	42
▪ THE DRVEERROR\$ RESERVED VARIABLE.....	43
MISCELLANEOUS RESERVED VARIABLES.....	43
▪ THE PRGNAME\$ RESERVED VARIABLE.....	43
▪ THE TIME RESERVED VARIABLE.....	44
▪ THE USR\$ RESERVED VARIABLE.....	44
▪ THE USRLVL RESERVED VARIABLE.....	44
▪ THE NETUP% RESERVED VARIABLE.....	44
SPECIAL PURPOSE RESERVED VARIABLES.....	45
▪ THE BUFFER\$ RESERVED VARIABLE.....	45
▪ THE TRIGGER\$ RESERVED VARIABLE.....	45
▪ THE TRIGGERDRV\$ RESERVED VARIABLE.....	46

LEGACY OBJECT RESERVED VARIABLES.....	46
■ THE TRIGGERMSG\$ RESERVED VARIABLE	46

FUNCTION REFERENCE 47

MATHEMATICAL FUNCTIONS.....	47
■ THE ABS FUNCTION.....	47
■ THE SQRT FUNCTION	47
■ THE SIN FUNCTION.....	47
■ THE COS FUNCTION.....	47
■ THE TAN FUNCTION.....	48
■ THE EXP FUNCTION.....	48
■ THE LN FUNCTION.....	48
■ THE LOG2 FUNCTION	48
■ THE LOG10 FUNCTION	48
■ THE MOD FUNCTION.....	49
■ THE RND FUNCTION.....	50
■ THE RNDDWN FUNCTION	50
■ THE RNDUP FUNCTION	50
STRING MANIPULATION FUNCTIONS	51
■ THE LEN FUNCTION.....	51
■ THE LEFT\$ FUNCTION	51
■ THE RIGHT\$ FUNCTION.....	51
■ THE MID\$ FUNCTION	51
■ THE POS FUNCTION.....	52
■ THE REGEXPOS FUNCTION.....	52
■ THE REGEXEND FUNCTION.....	52
STRING CONVERSION FUNCTIONS.....	53
■ THE CHR\$ FUNCTION	53
■ THE ASCII FUNCTION	53
■ THE SASCII FUNCTION.....	54
■ THE STR\$ FUNCTION	54
■ THE ISTR\$ FUNCTION	54
■ THE VAL FUNCTION.....	55
■ THE IVAL FUNCTION	55
■ THE HEXVAL FUNCTION.....	56
■ THE BINVAL FUNCTION.....	56
■ THE OCTVAL FUNCTION.....	56
■ THE BCDVAL FUNCTION.....	57
■ THE LOHIVAL FUNCTION.....	57
■ THE SLOHIVAL FUNCTION.....	57
■ THE HILOVAL FUNCTION.....	58
■ THE SHILOVAL FUNCTION.....	58
■ THE FMT\$ FUNCTION	58
■ THE HEXFMT\$ FUNCTION.....	59
■ THE HEXFMT2\$ FUNCTION.....	60
■ THE BINFMT\$ FUNCTION.....	60
■ THE OCTFMT\$ FUNCTION.....	60
■ THE BCDFMT\$ FUNCTION.....	61
■ THE LOHIFMT\$ FUNCTION.....	61
■ THE HILOFMT\$ FUNCTION.....	62
■ THE CCNV\$ FUNCTION	62
■ THE HCNV\$ FUNCTION	64
DATA DECODING/ENCODING FUNCTIONS	64
■ THE DECODE FUNCTION.....	64
■ THE DECODE\$ FUNCTION.....	64
■ THE DECODE% FUNCTION.....	65
■ THE DECODEREGEX\$ FUNCTION.....	65
■ THE ENCODE\$ FUNCTION.....	66

CHECKSUM FUNCTIONS	66
■ THE CHKSUM FUNCTION	66
■ THE CHKSUMLOHI FUNCTION	67
■ THE CHKSUMHILO FUNCTION	67
■ THE LRC\$ FUNCTION	68
■ THE LRCLOHI FUNCTION	68
■ THE LRCHILO FUNCTION	69
■ THE CRC16 FUNCTION	70
■ THE CRCCITT FUNCTION	70
■ THE CRC32 FUNCTION	71
■ THE CHKSUM\$ FUNCTION	71
■ THE PRNCHKSUM\$ FUNCTION	72
TIME AND DATE FUNCTIONS	73
■ THE TIME\$ FUNCTION	73
■ THE MKTIME FUNCTION	73
■ THE GMT FUNCTION	73
■ THE LCTIME FUNCTION	74
■ THE MON FUNCTION	74
■ THE MON\$ FUNCTION	74
■ THE MONAB\$ FUNCTION	74
■ THE DAY FUNCTION	75
■ THE YR FUNCTION	75
■ THE HRS FUNCTION	75
■ THE MINS FUNCTION	75
■ THE SECS FUNCTION	76
■ THE WKDAY FUNCTION	76
■ THE WKDAY\$ FUNCTION	76
■ THE WKDAYAB\$ FUNCTION	76
■ THE INTVMINS\$ FUNCTION	77
■ THE INTVHRS\$ FUNCTION	77
DIALOG BUTTON CALLBACK FUNCTIONS	77
■ THE LITEM\$ FUNCTION	77
■ THE LITEMEXISTS% FUNCTION	78
■ THE MAXLITEM FUNCTION	78
■ THE COUNTLITEMS FUNCTION	79
■ THE MITEM\$ FUNCTION	79
■ THE MITEMEXISTS% FUNCTION	80
■ THE MAXMITEM FUNCTION	80
■ THE COUNTMITEMS FUNCTION	81
FILE FUNCTIONS	81
■ THE FLEN FUNCTION	81
■ THE FPOS FUNCTION	82
REGISTER FUNCTIONS	82
■ THE REGNAME\$ FUNCTION	82
■ THE ONLOGSTR\$ FUNCTION	83
■ THE ONLOGSTR\$ FUNCTION	83
■ THE REGSTAT% FUNCTION	83
■ THE REGMASK% FUNCTION	83
■ THE REGHIDDEN% FUNCTION	84
■ THE REGERR% FUNCTION	84
■ THE BSTVAL% FUNCTION	84
■ THE BSTDLY FUNCTION	85
■ THE DIGVAL FUNCTION	85
■ THE DIGVAL FUNCTION	85
■ THE ANAVAL FUNCTION	85
■ THE ANAVAL FUNCTION	86
■ THE ANAHIGH FUNCTION	86
■ THE ANALOW FUNCTION	87
■ THE ANAMIN FUNCTION	87
■ THE ANAMAX FUNCTION	87
■ THE STRVAL\$ FUNCTION	88

SERIAL COMMUNICATION FUNCTIONS 88

- THE **DRVNDATA\$** FUNCTION88
- THE **DRVNEERROR** FUNCTION89
- THE **DRVNEERROR\$** FUNCTION89
- THE **DRVENABLED%** FUNCTION90
- THE **CMDENABLED%** FUNCTION90
- THE **DRVREADY%** FUNCTION90
- THE **SUSPENDED%** FUNCTION91

SERIAL DEVICE OBJECT FUNCTIONS 91

- THE **DRVOBJVAL** FUNCTION91
- THE **DRVOBJVAL\$** FUNCTION92
- THE **DRVOBJVAL%** FUNCTION93
- THE **DRVOBJGL** FUNCTION94
- THE **DRVOBJHIGH** FUNCTION94
- THE **DRVOBJLOW** FUNCTION95
- THE **DRVOBJMASK%** FUNCTION96
- THE **DRVOBJERR%** FUNCTION96

MISCELLANEOUS FUNCTIONS 97

- THE **USRPRV%** FUNCTION97
- THE **PVAR** FUNCTION98
- THE **PVAR\$** FUNCTION98
- THE **PVAR%** FUNCTION98

SPECIAL PURPOSE FUNCTIONS 99

- THE **DRVPRM** FUNCTION99
- THE **DRVPRM\$** FUNCTION99
- THE **DRVPRM** FUNCTION100
- THE **BUFFER** FUNCTION100
- THE **TRIGGERPRM** FUNCTION100
- THE **TRIGGERPRM\$** FUNCTION101
- THE **TRIGGERPRM%** FUNCTION101
- THE **RTSPRM\$** FUNCTION102

LEGACY OBJECT FUNCTIONS 102

- THE **MSGENABLED%** FUNCTION102
- THE **PARAM** FUNCTION103
- THE **PARAM\$** FUNCTION103
- THE **PARAM%** FUNCTION103

OBSOLETE FUNCTIONS 104

COMMAND REFERENCE **105**

FLOW CONTROL COMMANDS 105

- THE **GOTO** COMMAND105
- THE **GOSUB** COMMAND105
- THE **ON...GOTO** COMMAND105
- THE **ON...GOSUB** COMMAND106
- THE **RETURN** COMMAND106
- THE **IF** COMMAND106
- THE **ELSEIF** COMMAND107
- THE **ELSE** COMMAND107
- THE **ENDIF** COMMAND108
- THE **WHILE** COMMAND108
- THE **ENDDO** COMMAND109
- THE **REPEAT** COMMAND109
- THE **UNTIL** COMMAND109
- THE **FOR** COMMAND110
- THE **NEXT** COMMAND111
- THE **END** COMMAND111

USER MESSAGE COMMANDS 111

■	THE PROMPT COMMAND	111
■	THE INFO COMMAND	112
■	THE PRINT COMMAND	113
■	THE ERRMSG COMMAND	114
■	THE CONFIRM COMMAND	114
■	THE ASK COMMAND	115
	DIALOG COMMANDS	116
■	THE DIALOG COMMAND	116
■	THE DLGTITLE COMMAND	117
■	THE DLGTEXT COMMAND	117
■	THE DLGLINE COMMAND	118
■	THE STREDIT COMMAND	118
■	THE STREDIT0 COMMAND	119
■	THE PWEDIT COMMAND	121
■	THE PWEDIT0 COMMAND	122
■	THE NUMEDIT COMMAND	123
■	THE INTEDIT COMMAND	125
■	THE CHKBOX COMMAND	127
■	THE LIST COMMAND	127
■	THE LISTW COMMAND	129
■	THE LIST0 COMMAND	130
■	THE LISTW0 COMMAND	132
■	THE SLIST COMMAND	133
■	THE SLISTW COMMAND	134
■	THE SLIST0 COMMAND	136
■	THE SLISTW0 COMMAND	137
■	THE LITEM COMMAND	139
■	THE RDGRP COMMAND	139
■	THE RDGRP0 COMMAND	140
■	THE RDBTN COMMAND	141
■	THE MENU COMMAND	141
■	THE MENU0 COMMAND	143
■	THE MITEM COMMAND	144
■	THE BUTTON COMMAND	144
■	THE BUTTON0 COMMAND	145
■	THE CANCELBTN COMMAND	146
	DIALOG BUTTON CALLBACK COMMANDS	147
■	THE DLGERROR COMMAND	147
■	THE SETLITEM COMMAND	148
■	THE ADDLITEM COMMAND	148
■	THE DELLITEM COMMAND	149
■	THE CLRLITEMS COMMAND	150
■	THE SETMITEM COMMAND	150
■	THE ADDMITEM COMMAND	151
■	THE DELMITEM COMMAND	151
■	THE CLRMITEMS COMMAND	152
	FILE AND NETWORK CONNECTION COMMANDS	152
■	THE OPEN COMMAND	152
■	THE CONNECT COMMAND	153
■	THE CLOSE COMMAND	154
■	THE PRINT# COMMAND	155
■	THE INPUT# COMMAND	155
	If file_number is a file:	156
	If file_number is a network connection:	156
■	THE SETFPOS COMMAND	156
■	THE LIMITFLEN COMMAND	157
	REGISTER COMMANDS	157
■	THE SETREGNAME COMMAND	157
■	THE REVERTREGNAME COMMAND	158
■	THE SETONLOGSTR COMMAND	158

■	THE REVERTONLOGSTR COMMAND.....	158
■	THE SETOFFLOGSTR COMMAND.....	159
■	THE REVERTOFFLOGSTR COMMAND.....	159
■	THE SETBSTVAL COMMAND.....	159
■	THE SETBSTDLY COMMAND.....	160
■	THE SETDIGVAL COMMAND.....	160
■	THE SETANAVAL COMMAND.....	160
■	THE SETANAVALGL COMMAND.....	162
■	THE SETANAVALS COMMAND.....	163
■	THE SETANAVALSGL COMMAND.....	163
■	THE SETANAMIN COMMAND.....	164
■	THE CLRANAMIN COMMAND.....	164
■	THE SETANAMAX COMMAND.....	165
■	THE CLRANAMAX COMMAND.....	165
■	THE SETINDRANGE COMMAND.....	165
■	THE REVERTINDRANGE COMMAND.....	166
■	THE SETSTRVAL COMMAND.....	166
■	THE MASK COMMAND.....	167
■	THE UNMASK COMMAND.....	167
■	THE INTMASK COMMAND.....	168
■	THE INTUNMASK COMMAND.....	168
■	THE HIDE COMMAND.....	168
■	THE UNHIDE COMMAND.....	169
	SERIAL COMMUNICATION COMMANDS	169
■	THE GRAB COMMAND.....	169
■	THE RELEASE COMMAND.....	170
■	THE SENDCMD COMMAND.....	170
■	THE SENDSTR COMMAND.....	171
	Sending Custom Commands to Devices That Use Legacy Device Drivers	172
■	THE SENBIN COMMAND.....	172
	Sending Custom Commands to Devices That Use Legacy Device Drivers	173
■	THE DISABLEDRV COMMAND.....	173
■	THE ENABLEDRV COMMAND.....	174
■	THE DISABLECMD COMMAND.....	174
■	THE ENABLECMD COMMAND.....	175
■	THE SUSPEND COMMAND.....	176
■	THE RESUME COMMAND.....	176
	SERIAL DEVICE OBJECT COMMANDS	177
■	THE SETDRVBJVAL COMMAND.....	177
■	THE SETDRVBJVALGL COMMAND.....	179
■	THE SETDRVBJVALS COMMAND.....	180
■	THE SETDRVBJVALSGL COMMAND.....	181
■	THE MASKDRVOBJ COMMAND.....	182
■	THE UNMASKDRVOBJ COMMAND.....	183
	LOGGING COMMANDS	184
■	THE LOG COMMAND.....	184
■	THE LOGR COMMAND.....	184
■	THE LOGG COMMAND.....	184
■	THE LOGB COMMAND.....	185
■	THE LOGC COMMAND.....	185
■	THE LOGM COMMAND.....	185
■	THE LOGY COMMAND.....	185
■	THE FILELOG COMMAND.....	186
■	THE FILELOGR COMMAND.....	186
■	THE FILELOGG COMMAND.....	186
■	THE FILELOGB COMMAND.....	187
■	THE FILELOGC COMMAND.....	187
■	THE FILELOGM COMMAND.....	187
■	THE FILELOGY COMMAND.....	187
	DATA ENCODING/DECODING COMMANDS	188

■	THE PARSEDEC COMMAND	188
■	THE PARSEREGEX COMMAND	188
■	THE SKIPDEC COMMAND	189
■	THE SKIPREGEX COMMAND	190
■	THE APPENDSTR COMMAND	190
■	THE APPENDCSTR COMMAND	191
■	THE APPENDHEX COMMAND	193
■	THE APPENDENC COMMAND	193
	MISCELLANEOUS COMMANDS	194
■	THE SETPVAR COMMAND	194
■	THE DELAY COMMAND	194
■	THE CALL COMMAND	195
■	THE DRVCALL COMMAND	196
■	THE CALLRMT COMMAND	196
■	THE RUN COMMAND	198
■	THE DRVRUN COMMAND	199
■	THE RUNRMT COMMAND	200
■	THE LAUNCH COMMAND	201
■	THE STOPNET COMMAND	201
■	THE STARTNET COMMAND	202
	SPECIAL PURPOSE COMMANDS	202
■	THE SABUSREPLY COMMAND	202
■	THE SABUSERERROR COMMAND	203
■	THE RTSEND COMMAND	203
■	THE RTSERROR COMMAND	204
	LEGACY OBJECT COMMANDS	204
■	THE SENDREPLY COMMAND	204
■	THE DISABLEMSG COMMAND	205
■	THE ENABLEMSG COMMAND	205
■	THE SETPARAM COMMAND	206
	OBSOLETE COMMANDS	206

APPENDICES

207

APPENDIX A: ALPHABETICAL LIST OF KEYWORDS	207
---	-----

A	207
B	207
C	207
D	207
E	207
F	207
G	207
H	207
I	207
J	207
K	207
L	207
M	208
N	208
O	208
P	208
Q	208
R	208
S	208
T	208
U	208
V	208
W	208
X	209
Y	209
Z	209

APPENDIX B: LIST OF ERROR MESSAGES	210
▪ Array has wrong number of dimensions	210
▪ Array name expected	210
▪ Array subscript expected.....	210
▪ Array too large	210
▪ Assignment operator expected.....	210
▪ Bad C-style string	210
▪ Bad expression list termination.....	210
▪ Bad hex value string.....	210
▪ Bad regular expression.....	210
▪ Binary operator expected	210
▪ Boolean decoder expected	210
▪ Boolean expected	210
▪ Boolean Variable expected	210
▪ Byte (-128 to 127 or 0 to 255) expected	210
▪ Command or assignment expected	210
▪ Device driver object is wrong type	211
▪ Device driver parameter is wrong type.....	211
▪ Dialog list has no item with that number	211
▪ Dialog list item command or function outside dialog subroutine	211
▪ Dialog list item command or function variable not used in dialog	211
▪ Dialog menu has no item with that number	211
▪ Dialog menu item command or function outside dialog subroutine	211
▪ Dialog menu item command or function variable not used in dialog.....	211
▪ Dialog object is not a list	211
▪ Dialog object is not a menu	211
▪ Digit expected.....	211
▪ Division by zero.....	211
▪ "DLGERROR" outside dialog subroutine	211
▪ "DLGERROR" variable not used in dialog	211
▪ "DO" without "WHILE"	211
▪ Duplicate "ELSE"	211
▪ Duplicate file number	212
▪ Duplicate line number.....	212
▪ Duplicate port	212
▪ Duplicate program argument	212
▪ "ELSE" without "IF"	212
▪ "ENDDO" without "DO"	212
▪ "ENDIF" without "IF"	212
▪ Expression is not a variable	212
▪ Expression is not an array	212
▪ File not open	212
▪ File was opened for reading only.....	212
▪ File was opened for writing only	212
▪ Identifier is not a function.....	212
▪ Identifier is not an array.....	212
▪ Illegal array subscript.....	212
▪ Illegal character.....	212
▪ Illegal command for device driver program	212
▪ Illegal digit.....	212
▪ Illegal function for device driver program.....	213

▪ Illegal line number	213
▪ Illegal value.....	213
▪ Inappropriate command	213
▪ Inappropriate function.....	213
▪ "INPUT#" needs length for network connection	213
▪ Integer (-2,147,483,648 to 2,147,483,647) expected	213
▪ Maximum number of instructions exceeded.....	213
▪ Misplaced array subscript	213
▪ Misplaced binary operator	213
▪ Misplaced command	213
▪ Misplaced command separator	213
▪ Misplaced decimal point.....	213
▪ Misplaced "ELSE"	213
▪ Misplaced "ELSEIF"	213
▪ Misplaced "ENDDO"	213
▪ Misplaced "ENDIF"	214
▪ Misplaced exponent	214
▪ Misplaced line continuation character ("\")	214
▪ Misplaced "LITEM"	214
▪ Misplaced "MITEM"	214
▪ Misplaced parameter separator (",")	214
▪ Misplaced print list separator (";").....	214
▪ Misplaced "RDBTN"	214
▪ Misplaced remark	214
▪ Misplaced sign	214
▪ Misplaced "UNTIL"	214
▪ Misplaced value	214
▪ Missing array name.....	214
▪ Missing "DO"	214
▪ Missing "ENDDO"	214
▪ Missing "ENDIF"	214
▪ Missing "GOTO" or "GOSUB"	214
▪ Missing "THEN"	214
▪ Missing "TO"	215
▪ "NEXT" without "FOR"	215
▪ Network connection has no size or position	215
▪ Number expected	215
▪ Numerical decoder expected.....	215
▪ Numerical variable expected.....	215
▪ Overflow	215
▪ Parameter list expected	215
▪ Port not grabbed.....	215
▪ Positive integer (1 to 4,294,967,295) expected.....	215
▪ Register is wrong type	215
▪ "RELEASE" without "GRAB"	215
▪ "RETURN" without "GOSUB"	215
▪ Second "CANCELBTN"	215
▪ Second "GRAB"	215
▪ Second "SABUSREPLY" or "SABUSEROR"	215
▪ Single statement after "ELSEIF"	216
▪ "STEP" without "FOR"	216

▪ String contains non-printable characters.....	216
▪ String decoder expected.....	216
▪ String expected.....	216
▪ String Variable expected.....	216
▪ Superfluous array subscript.....	216
▪ Syntax error.....	216
▪ "THEN" without "IF".....	216
▪ "TO" without "FOR".....	216
▪ Too few arguments.....	216
▪ Too few indices.....	216
▪ Too few parameters.....	216
▪ Too many arguments.....	216
▪ Too many indices.....	216
▪ Too many parameters.....	216
▪ Trigger object parameter is wrong type.....	216
▪ Type mismatch.....	216
▪ Unexpected end of line.....	217
▪ Unknown application.....	217
▪ Unknown decoder.....	217
▪ Unknown device command.....	217
▪ Unknown device driver.....	217
▪ Unknown device driver object.....	217
▪ Unknown device driver parameter.....	217
▪ Unknown device message.....	217
▪ Unknown device reply.....	217
▪ Unknown device response.....	217
▪ Unknown encoder.....	217
▪ Unknown line number.....	217
▪ Unknown parameter.....	217
▪ Unknown register.....	217
▪ Unknown SCL program.....	217
▪ Unknown serial port.....	217
▪ Unknown trigger object parameter.....	217
▪ Unmatched left parenthesis ("").....	217
▪ Unmatched right parenthesis ("").....	217
▪ Unmatched start array subscript character ("[".....	218
▪ Unmatched string delimiter (double quotes).....	218
▪ Unsigned integer (0 to 4,294,967,295) expected.....	218
▪ "UNTIL" without "REPEAT".....	218
▪ Value expected.....	218
▪ Variable is not an array.....	218
▪ Variable name expected.....	218
▪ Wrong "NEXT" variable.....	218

CONTACT INFORMATION **219**

U.P.M.A.C.S. COMMUNICATIONS INC.....	219
--------------------------------------	-----

THE SCL PROGRAMMING LANGUAGE

Introduction

SCL is a control language based on the BASIC programming language.

Each line of an SCL program consists of an optional line number, followed by a series of commands or assignments separated by colons (":"). SCL keywords are not case sensitive, i.e. "print" is treated the same as "PRINT" and "PRinT".

Keywords are words that SCL knows. Some keywords are built into the language, like GOTO and CHR\$. Others, you define yourself by creating variables.
All keywords must be unique.

Here are examples for lines of SCL code:

```
10 PRINT "Hello, World"  
A=10 : GOTO 10
```

Long lines of code can be broken into several lines using a backslash ("\") as a continuation character:

```
10 PRINT \  
    "Hello, World"
```

You cannot place the continuation character in the middle of a string, number, or keyword.

LANGUAGE COMPONENTS

Remarks

You can place remarks and comments in an SCL program using the `REM` statement. SCL ignores everything after the `REM` statement. If the `REM` statement follows a command or assignment, separate the two with a colon.

Here are some examples of `REM` statements:

```
REM This is a comment ignored by SCL. It is intended for the user
GOTO 1:REM *** Go back to the beginning. ***
```

Literal Values

Literal values are values that are hard-coded into your SCL program. There are two types of values you can specify in a program:

Numbers

Numbers are just that: numbers. You can either enter decimal numbers, or you can enter hexadecimal, octal, or binary numbers by placing the *base prefix* before the number. The base prefix for hexadecimal numbers is the `$` sign, the base prefix for octal numbers is the `&` sign, and the base prefix for binary numbers is the `%` sign. Hexadecimal, octal, and binary numbers cannot contain fractions or exponents.

<p>Examples of decimal numbers:</p> <pre>10 12.4 6.21E+23</pre> <p>(the third number is $6.21 \cdot 10^{23}$)</p> <p>Examples of hexadecimal numbers:</p> <pre>\$2F \$0D04 \$3</pre>	<p>Examples of binary numbers:</p> <pre>%1001 %01 %1111001</pre> <p>Examples of octal numbers:</p> <pre>&35407 &220 &00503</pre>
---	--

Strings

Though SCL strings can contain text or binary data, string literals only support text. Enclose the text in double quotes:

```
"This is a string literal in double quotes"
```

Constants

Constants are keywords that represent certain fixed values. You can never change the value of a constant, and you cannot define any constants yourself. SCL has six built-in constants. Each constant has one of three types: numerical, string, or Boolean type.

Name	Type	Value
TRUE%	Boolean	true
FALSE%	Boolean	false
PI	numerical	3.14159265359
RET\$	string	carriage return (Hex \$0D)
TAB\$	string	tab character (Hex \$08)
QUT\$	string	double quotes (" ")

Boolean values are values that can be either true or false. They are used in `IF` statements, for example.

Reserved Variables

Reserved variables are keywords that the SCL interpreter sets to certain values representing important information. Like constants, You cannot change the value of a reserved variable. Unlike constants, however, the value of a reserved variable may be changed by the SCL interpreter while the program is running. SCL has seven reserved variables. Each reserved variable has one of three types: numerical, string, or Boolean type.

Examples of reserved variables include the name of the program being executed, the current time, and the name of the user that is currently logged on.

User Variables

User variables are keywords that you define yourself for your own use. A user variable is automatically created and set to a default value (see below) the first time you use it. You can change the value of a user variable by using an assignment, or by using a command that changes the value, like the `INPUT#` command.

User variables can have one of three types: numerical, string, or Boolean type.

The name of a user variable consist of any combination of letters, digits, and the underscore character (" _"), followed by an optional *type suffix*. The first character in a variable name cannot be a digit.

SCL determines the type of the user variable by its suffix:

- *No suffix*: numerical variable
- \$: string variable
- %: Boolean variable

Don't separate the type suffix from the rest of the name by spaces.

Examples of numerical variables:	Examples of string variables:	Examples of Boolean variables:
A B2 POWER_IN_DBW	A\$ ERROR_MESSAGE\$ PARAM_01\$	A% REMOTE_MANUAL% SWITCH_03_ON%

SCL initializes user variables to the following values:

- Numerical: 0
- String: empty string
- Boolean: false

Variables specified by program arguments are automatically created and set to values defined by the argument, and are not initialized to the default values. Similarly, variables specified in variable command parameters of SABus commands are automatically created and set to the value parsed from the SABus request packet.

Arrays

Arrays are a special kind of user variables. They contain a whole set of values, instead of just one.

Indices

Each individual value stored in an array is called an *element*. Each element has its own *index* or *indices*. An index is simply an integer number (positive, negative, or zero).

How many indices are needed to specify an element depends on the number of *dimensions* the array has. A one-dimensional array needs one index, a two dimensional array needs two indices, etc. SCL determines the number of dimensions of an array by the number of indices you specify the first time you use an element.

Once you have used an array, you can never change the number of dimensions it has.

Creating Arrays

You create an array simply by using one of its elements. SCL will determine the number of dimensions the array has from the number of indices you specify. The same rules apply for array names as for user variable names.

The elements of an array are initialized to the same default values as user variables.

Accessing Elements

You can use an array element anywhere you would use a regular user variable. To specify an individual element, put the indices in square brackets after the array name. If the array has more than one dimension, separate the indices by commas.

To access the element with the index 2 of the array MESSAGE\$, for example, you could write any of the following:

```
MESSAGE$[2]
MESSAGE$ [ 1+1 ]
MESSAGE$[1/SIN(PI/3)]
```

If you happen to have a variable A with value 6, you could also write:

```
MESSAGE$[A/3]
```

You can use any numerical expression as an array index. Arrays are theoretically infinite in size, but they must fit into available memory, of course.

To access the element with indices x, y and z in a three-dimensional array called FLUX, write the following:

```
FLUX[X, Y, Z]
```

Functions

Functions are keywords used to perform predefined calculations or to retrieve information. SCL has a multitude of pre-defined functions, described in the function reference. Each function takes one or more parameters (numbers, strings, or Boolean values), and returns a certain value that depends on them.

Examples of functions include the square root function `SQRT`, and the function `CHKSUM$`, which calculates the modulo 256 checksum of a string.

There are numerical functions, string functions and Boolean functions, depending on the type of the value they return. You can use a function anywhere you would use a constant or a literal value of the same type.

To use a function, specify its parameters in parentheses after its name. If the function takes more than one parameter, separate them by commas.

You can use any expression as a function parameter, as long as the result has the required type.

Examples:

```
SQRT(2)
LEFT$(A$, 4)
LN(INPUT_VALUE/10)
```

Mathematical Expressions

SCL has a fully qualified mathematical expression parser. This means that you can enter complex equations directly into SCL code. SCL fully supports nested brackets ("()"), as well as the following operators:

Operator	Result Type	Function
+	numerical	adds two numbers
+	string	concatenates two strings (attaches them together)

-	numerical	subtracts two numbers
/	numerical	divides two numbers
*	numerical	multiplies two numbers
^	numerical	raises one number to the power of the other (2^3=8)
AND	Boolean	logical and
AND	numerical	bitwise and
OR	Boolean	logical or
OR	numerical	bitwise or
XOR	Boolean	logical exclusive or
XOR	numerical	bitwise exclusive or
=	Boolean	determines if two numbers, strings, or Booleans are equal
<>	Boolean	determines if two numbers, strings, or Booleans are not equal
<, >, <=, >=	Boolean	compares two numbers or strings
NOT	Boolean	inverts a Boolean (turns true into false and vice versa)

Examples of numerical expressions:

```
1.25 + 3*A + 1.2*A^2 + 3.2*A^3 + 4.4*A^4
SQRT(A^2+B^2)
(-b+SQRT(b^2-4*a*c))/(2*a)
```

Examples of string expressions:

```
A$+CHKSUM$(A$)+CHR$( $0D)
>Your name is :"+NAME$
"Publius"+TAB$+"Vergilius"+TAB$+"Maro"+RET$
```

Examples of Boolean expressions:

```
REGSTAT%("Remote") AND REGSTAT%("Manual")
NETUP% OR OVERRIDE=1
A<2 OR (A$=B$ AND NOT F%)
```

Operator Priority

Operators are evaluated highest to lowest priority. This means that $1+2*3$ is interpreted as $1+(2*3)$ rather than $(1+2)*3$. Operators with the same priority are evaluated left to right.

Here is a list of all operators from highest to lowest priority:

```
^
*, /
+, -
NOT
=, <>, <, >, <=, >=
AND
OR, XOR
```

Assignments

An assignment lets you assign a value to a user variable or array element. You place an assignment by itself on a line of code, or separated by colons (":") from other assignments or commands.

An assignment is a user variable or array element followed by an equal sign and an expression of the same type as the variable. SCL will set the value of the user variable or array element to the result of the expression.

Commands

Commands are keywords that let you perform certain actions, like setting the value of a register, asking the user for input, or jumping to a different place in the program. You place a command followed by all its parameters on a single line of code, separated by colons (":") from other commands or assignments.

Commands can take arguments. Some commands do not take any arguments, some commands take expressions, others variables, still others both. Some commands can have optional arguments, or a variable number of arguments.

Command arguments are not enclosed in parentheses. They simply follow the command, separated by commas. Here are some examples:

```
GOTO 20  
SETDIGVAL "HPA 1 Frequency Channel", 12
```

PROGRAMMING METHODS

Program Arguments

SCL programs can have a number of arguments. The object that runs the program, e.g. a control button or processor source, can supply any number of arguments. Each argument specifies a user variable name and a corresponding value. A variable with the given name is created and initialized with the value. This allows you to reuse the same program for similar tasks, and prevents repetitive writing of many programs with small differences.

Conditional Statements

SCL conditional statements come in two flavours:

Single Command Conditions

A single command condition allows you to specify a command or assignment that will only be executed if a certain Boolean expression (the *condition*) is true. A single command condition uses the `IF` and `THEN` keywords:

```
IF condition% THEN command
```

The command will only be executed if `condition%` evaluates to true.

IF-THEN-ENDIF Blocks

An `IF-THEN-ENDIF` block allows you to specify a block or a series of blocks of commands and assignments that will only be executed if a number of Boolean expressions are true. The simplest `IF-THEN-ENDIF` block is constructed using only the `IF`, `THEN`, and `ENDIF` keywords:

```
IF condition% THEN  
  command 1  
  command 2  
  command 3  
  etc.  
ENDIF
```

The commands will only be executed if `condition%` evaluates to true.

Note how the first command is not on the same line as the `THEN` keyword. If you place them on the same line, you must separate them using a colon, or the parser will interpret it as a single command condition.

You can also specify another block that is to be executed if `condition%` is false by using the `ELSE` keyword:

```
IF condition% THEN  
  command 1
```

```
    command 2
    command 3
    etc.
ELSE
    command 4
    command 5
    command 6
    etc.
ENDIF
```

The commands 1, 2, 3, etc. will be executed if `condition%` evaluates to true. If `condition%` is false, commands 4, 5, 6, etc. will be executed instead.

You can specify multiple conditions by using the `ELSEIF` keyword:

```
IF condition_1% THEN
    command 1
    command 2
    command 3
    etc.
ELSEIF condition_2% THEN
    command 4
    command 5
    command 6
    etc.
ENDIF
```

If `condition_1%` is true, commands 1, 2, 3, etc. will be executed.

If `condition_1%` is false, but `condition_2%` is true, commands 4, 5, 6, etc. will be executed.

You can have any number of `ELSEIF` blocks within an `IF-THEN-ENDIF` block. You can also combine `ELSEIF` and `ELSE` blocks:

```
IF condition_1% THEN
    command 1
    command 2
    command 3
    etc.
ELSEIF condition_2% THEN
    command 4
    command 5
    command 6
    etc.
ELSEIF condition_3% THEN
    command 7
    command 8
    command 9
    etc.
ELSE
    command 10
    command 11
    command 12
```

etc.
ENDIF

If `condition_1%` is true, commands 1, 2, 3, etc. will be executed.

If `condition_1%` is false, but `condition_2%` is true, commands 4, 5, 6, etc. will be executed.

If `condition_1%` and `condition_2%` are false, but `condition_3%` is true, commands 7, 8, 9, etc. will be executed.

If all conditions are false, commands 10, 11, 12, etc. will be executed.

Line Numbers, Jumps, and Subroutines

Line Numbers

You can jump to a different line in an SCL program by using line numbers. A line number is simply a number that you place at the beginning of a line:

```
100 A=7
```

This line is line number 100. The line numbers need not be in order, and not every line has to have a line number. The following code will execute error-free:

```
20 PRINT "Hello!"  
PRINT "My name is Johnny."  
3000 GOTO 0: REM This line number is never used.  
  
0 PRINT "We have reached line number zero!"  
PRINT "Let's start over"  
GOTO 20: REM Go back to the beginning.
```

Each line number must be a unique integer between 0 and 4,294,967,295. You cannot use the same line number twice.

Jumps

To jump to a line with a certain line number, use the `GOTO` command:

```
      GOTO 100: REM Jump to line 100  
      PRINT "This line is never executed"  
100 END
```

You can use any numerical expression in a `GOTO` statement, as long as it results in an existing line number:

```
      A=5  
      GOTO 10*A  
50 PRINT "Hurray!"
```

Subroutines

Subroutines are not specially marked in SCL, and do not have names. You jump to a subroutine using the `GOSUB` (go to subroutine) command. The subroutine will end once a `RETURN` statement is encountered. After the subroutine has finished executing, SCL will continue with the statement after the `GOSUB` statement.

```
REM Main program body
PRINT "Starting up..."
PRINT "Trying the subroutine..."
GOSUB 10000: REM Call the subroutine
PRINT "Returned from subroutine!"
END: REM if this end were not here
      REM execution would continue into the subroutine.

REM Subroutine
10000 PRINT "I'm in the subroutine!"
      RETURN :REM return to where we came from
```

You can call subroutines from various places in the program. The `RETURN` command will always return to the statement right after the last `GOSUB`. You can also call subroutines from within subroutines.

If SCL encounters a `RETURN` when not within a subroutine, an error is generated.

Multibranching

SCL has a special way of jumping called multibranching. Using the `ON...GOTO` and `ON...GOSUB` commands, you can jump to different line numbers of subroutines depending on the value of a numerical expression. You can specify line numbers to jump to for values of 1, 2, 3, 4, etc.. An `ON...GOTO` multibranch looks like this:

```
ON index GOTO 100, 110, 120, 1000
```

This command will jump to line number 100 if `index` is 1, to line number 110 if `index` is 2, to line number 120 if `index` is 3, or to line number 1000 if `index` is 4. for all other values of `index`, the command will not jump at all, but continue with the next line.

You can specify any number of line numbers.

The `ON...GOSUB` command works just like the `ON...GOTO` command, but the line numbers are treated as subroutines. Use the `RETURN` command to return from the subroutine.

Loops

WHILE-DO Loops

You can tell SCL to execute a block of commands and assignments repeatedly as long as a Boolean expression is false. You do this by using a `WHILE-DO` loop. `WHILE-DO` loops come in two flavours: Single command loops and `WHILE-DO-ENDDO` blocks.

A single command loop looks like this:

```
WHILE continue_contition% DO command
```

A WHILE-DO-ENDDO block looks like this:

```
WHILE continue_contition% DO  
  command 1  
  command 2  
  command 3  
  etc.  
ENDDO
```

Note how the first command in the WHILE-DO-ENDDO block is not on the same line as the DO keyword. If you place them on the same line, you must separate them using a colon, or the parser will interpret it as a single command loop.

In either case, the program checks `continue_condition%` when it reaches the WHILE keyword. If `continue_condition%` is true, the command or block of commands is executed, and the program jumps back to the WHILE statement to test `continue_condition%` again. If `continue_condition%` is false, the command or block is skipped, and execution continues after it.

These two things to distinguish WHILE-DO loops from REPEAT-UNTIL loops:

- The block may not be executed *at all*.
- The loop is aborted when the condition is *false*.

REPEAT-UNTIL Loops

You can tell SCL to execute a block of commands and assignments repeatedly until a Boolean expression is true. You do this by using a REPEAT-UNTIL loop:

```
REPEAT  
  command 1  
  command 2  
  command 3  
  etc.  
UNTIL stop_condition%
```

Every time the program reaches the UNTIL statement, it evaluates `stop_condition%`. If `stop_contition%` is false, it jumps back to the REPEAT statement. If `stop_condition%` is true, it goes on to the next command or assignment after the UNTIL.

These two things to distinguish REPEAT-UNTIL loops from WHILE-DO loops:

- The block is executed *at least once*.
- The loop is aborted when the condition is *true*.

FOR-NEXT Loops

A FOR-NEXT loop allows you to specify a block that will be executed a specific number of times. This is done by "counting" from one number to another. You must specify a numerical variable or array element that the program uses to count. a simple FOR-NEXT loop looks like this:

```
FOR counter = initial_value TO final_value  
    command 1  
    command 2  
    command 3  
    etc.  
NEXT counter
```

(You can also just write NEXT instead of NEXT counter)

The program will start "counting" by setting counter to initial_value. It will then execute the block of commands. Once it reaches the NEXT statement, it will add 1 to counter and execute the block again.

The block will be executed once with counter set to each value between initial_value and final_value, inclusive. You can, of course, access counter within the block like any other variable.

If final_value is less than initial_value, the program will count backwards.

initial_value and final_value do not have to be integers. You can count from -3.2 to 7.2, if you need to. If you specify a final_value that will not be reached exactly, the program will stop counting before it reaches final value:

```
FOR counter = 2.1 TO 8.5
```

This will count 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1, and then stop.

If you need to count in increments other than 1, use the step command to specify the increment:

```
FOR counter = initial_value TO final_value STEP step_size  
    command 1  
    command 2  
    command 3
```

etc.

NEXT

If `step_size` is positive, the program counts forward. If `step_size` is negative, it counts backward. If the sign of `step_size` would cause the program to count in the wrong direction (i.e. away from `final_value` instead of towards it), the block of commands is executed once with counter set to `initial_value`. A step size of 0 is not allowed.

After the loop is finished, counter will be one `step_size` *beyond* the value it had when the loop last executed. In the example used above:

```
FOR counter = 2.1 TO 8.5
```

counter will have the value 9.1 after the loop is done.

If `step_size` is 0, the parser generates an error.

If the variable specified in the `NEXT` statement is not the same as that specified in the last `FOR` statement, the parse generates an error.

Messages to the User

When to Use Which Command

SCL has several ways in which you can communicate with the user. You can display three kinds of messages:

- `INFO`, `PRINT`, and `ERRMSG` dialogs have only an **OK** button
- `CONFIRM` dialogs also have a **Cancel** button
- `ASK` dialogs have a **Yes** and a **No** button

The title of the message window will be the program title.

The messages are constructed using an internally maintained buffer. This buffer is filled with text by using the `PROMPT` command.

When you use any of the message commands, the output buffer will be shown in a message window. Once it has been displayed, the buffer will be emptied.

The `PRINT`, `INFO`, and `ERRMSG` commands allow you to specify additional expressions to be placed in the buffer. The `CONFIRM` and `ASK` commands only use the text already in the buffer.

Example of an **INFO** message:

```
PROMPT "The world will end ";
INFO "in ";2*15;" days."
```

displays the following window:



Example of a **PRINT** message:

```
PROMPT "Sumer is icumen in"
PROMPT "Lhude sing cu-cu"
PRINT
```

displays the following window:



Example of an **ERRMSG** message:

```
ERRMSG "I'm sorry, Dave,";\
" I can't do that."
```

displays the following window:



Example of an **ASK** message:

```
PROMPT "Will you marry me?"
ASK
```

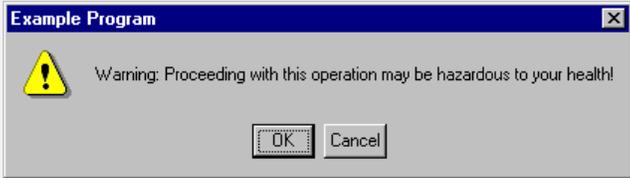
displays the following window:



Example of a **CONFIRM** message:

```
PROMPT "Warning: Proceeding with this operation may ";
PROMPT "be hazardous to your health!"
CONFIRM
```

displays the following window:



When to Use Which Command

Use the following guidelines for selecting a message command:

- If the user is waiting for a specific message as the result of something he did, use the **INFO** command. For example, use the **INFO** command to inform the user that a command was completed successfully.
- If a command from the user cannot be executed because of a condition that will occur during normal operation, use the **PRINT** command. For example, use the **PRINT** command to inform the user that a switch cannot be switched because it is in local mode, or because it is busy.

- If a user is waiting for a message, but the message you display is different from the one he is waiting for, also use the `PRINT` command. For example, use the `PRINT` command to tell the user that a smart switch switched to a secondary backup unit instead of the main backup.
- If a user is not waiting for a message at all, but the message is about something that occurs during normal operation, use the `PRINT` command as well. For example, use the `PRINT` command to tell the user that an air conditioning filter needs to be replaced.
- If an action could not be performed because of a condition that should not occur during normal operation, inform the user using the `ERRMSG` command. For example, use the `ERRMSG` command if a command failed because a piece of equipment does not respond, if a configuration option that should be installed is not installed, if an equipment firmware number is wrong, or if a required file cannot be found.
- If you detect a condition that should be impossible, use the `ERRMSG` command. For example, use the `ERRMSG` command to inform the user that ganged switches have different positions.

Note: An alarm should be considered a condition that occurs during normal operation. If a frequency cannot be set because a unit is in alarm, or if a switch cannot be switched because of an alarm, use the `PRINT` rather than the `ERRMSG` command.

- If you need to confirm a user command (display an "are you sure?"-message), use the `CONFIRM` command.
- If you need to inform the user about possible consequences of an action, and the user might change his mind about executing a command due to those consequences, also use the `CONFIRM` command. For example, use the `CONFIRM` command to tell the user that a data rate change on a modem carrying traffic will cause a loss of carrier.
- Anytime you want to give the user a choice to proceed with an operation, or to abort it, use the `CONFIRM` command.
- If something can be done in two equally valid ways, use the `ASK` command to let the user decide. For example, use the `ASK` command to determine whether a user wants an HPA automatically to maintain an output power he just entered, or not.
- If there is something that you feel the user might want to do in addition to the command he executed, use the `ASK` command to determine whether it should be done. For example, use the `ASK` command to determine whether the user wants to remove the RF inhibit from an HPA that he just switched on line.

Message commands are not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

Dialogs

How Dialogs Work

SCL supports fully featured dialogs to allow you to interact with the user.

To display a dialog, you must first construct it using the dialog item commands. SCL arranges the items from top to bottom in the order you added them. Buttons are placed on the right of the dialog, also from top to bottom in the order you added them. If you don't specify any buttons, SCL adds an OK and a Cancel button for you. Once you have constructed the dialog you display it using the `DIALOG` command.

Each dialog item has a variable associated with it. The item gets initialized to the value that the variable has when you use the `DIALOG` command, and the variable gets set to reflect the value the user entered in the item when he presses a button.

Usually, the dialog title will be the title of the program. You can change the title, however, by using the `DLGTITLE` command.

You can specify a result variable and a callback line number in the `DIALOG` command to determine the button the user pressed, and to check the values the user entered for validity before the dialog is closed. If the user entered invalid data, you can tell him so using a message command, and you can take him back to the item where invalid data was entered.

Once you use the `DIALOG` command, the dialog is deleted, and any new dialog commands you use will be applied to a new dialog.

Dialog Items

The following commands create items in a dialog. The items are placed in the dialog, from top to bottom, in the same order as you add them.

- To add explanatory text, use the `DLGTEXT` command.
- To place a horizontal line between items to group them, use the `DLGLINE` command
- To add a text entry field, use the `STREDIT`, `STREDIT0`, `PWDEDIT`, or `PWDEDIT0` command
- To add an entry field for numbers, use the `NUMEDIT` or `INTEDIT` command
- To add a check box, use the `CHKBOX` command
- To add a list box, use the `LIST`, `LISTW`, `LIST0`, `LISTW0`, `SLIST`, `SLISTW`, `SLIST0`, or `SLISTW0` command
- To add items to a list box, use the `LITEM` command
- To add a group of radio buttons, use the `RDGRP` or `RDGRP0` command
- To add radio buttons to a group, use the `RDBTN` command

- To add a popup menu (combo box) use the `MENU` or `MENU0` command
- To add items to a menu, use the `MITEM` command

All items except for `DLGTEXT` and `DLGLINE` items have a variable associated with it. Initially, the item is updated to reflect the value of the variable. When the user presses a button, the variable's value is set to reflect the user entry.

If the user presses the Cancel button, however, the variable values are reverted to their original values and all user entry is ignored.

Dialog Buttons

The following commands create buttons in a dialog. The buttons are placed along the right of the dialog, from top to bottom, in the same order as you add them.

- To add a button to the buttons on the right of the dialog, use the `BUTTON` or `BUTTON0` command
- To add a cancel button use the `CANCELBTN` command

Each button has a button number. The cancel button always has button number 0, all other buttons are numbered from one up, in the order you add them. You can determine which button the user pressed by the button number.

Buttons you add using the `BUTTON0` and `CANCELBTN` commands are always enabled and can always be pressed. Buttons you add using the `BUTTON` command are disabled and cannot be pressed if:

- any entry field created using the `STREDIT`, `PWDEDIT`, `NUMEDIT`, or `INTEDIT` command has no text in it
- any list box created using the `LIST`, `LISTW`, `SLIST` or `SLISTW` command has no selection
- any radio group created using the `RDGRP` command has no button pressed
- any menu created using the `MENU` command has no selection

Items created using the `STREDIT0`, `PWDEDIT0`, `LIST0`, `LISTW0`, `SLIST0`, `SLISTW0`, `RDGRP0`, and `MENU0` command do not affect any buttons.

If you do not specify any buttons, SCL will automatically add an OK and a Cancel button. The OK button will behave as if it was added using the `BUTTON` command, and will only be available under the conditions stated above.

The Result Variable

The `DIALOG` command allows you to specify a result variable. If you specify a result variable, the variable will contain the button number of the button the user pressed to close the dialog.

If you do not specify a result variable, the program will be aborted if the user presses the cancel button. If the user presses any other button, the program will continue. If you have more than one button, there is no way of telling which button the user pressed without specifying a result variable.

Caution: Please be aware that even if your dialog has no Cancel button, the result variable may be 0. The result variable is always 0 if the program was invoked from a remote computer, and communications are lost while the dialog is up, or if the program is called while the station is being closed or U.P.M.A.C.S. is quitting. You should always provide proper handling for a result variable value of 0.

The Button Callback

If you specify a result variable, you can also specify a button callback line number.

Whenever the user presses a button other than the Cancel button, SCL jumps to the line number as if it encountered a `GOSUB` command. You can look at the result variable to see what button the user pressed. All the variables associated with the items are updated to reflect the user's entries.

You return from the button callback using the `RETURN` command.

If you want the dialog to be closed when you return, do not modify the result variable. If you want the dialog to stay up, set the result variable to 0. If you set the result variable to any other value but zero, the dialog will be closed, and the result variable will retain that value when the program continues after the `DIALOG` statement.

If you set the result variable to 0, all dialog items will be updated to reflect any changes you made in their variables, and the dialog will stay up. The button callback will be called again once the user presses another button.

The button callback is not called if the user presses the Cancel button.

You can use the callback in two ways:

- **Perform validation of data the user entered**

If you find that the user entered values which are acceptable, just use the `RETURN` command. The dialog will be closed.

If you find that the user entered a value that is not acceptable, pop up a message using the `PRINT` command, informing the user of which field is not acceptable, and why. Then use the `DLGERROR` command to highlight the problem field in the dialog, set the result variable to 0, and use the `RETURN` command. The dialog will stay up, and the user can try to enter valid values.

- **Perform an action that does not close the dialog**

If you want to have a button in your dialog that performs an action but does not close the dialog, you can implement it using the button callback.

Simply set the result variable to 0 if the special button was pressed, and perform any actions that the button press is supposed to produce. You can modify the variables associated with the dialog items to change the content of the dialog, use the `SETLITEM`, `ADDLITEM`, `DELLITEM` and `CLRLITEMS` to change the content of lists, use the `SETMITEM`, `ADDMITEM`, `DELMITEM` and `CLRMITEMS` to change the content of menus, or perform any other actions that might be necessary. Then use the `RETURN` command to go back to dialog processing.

You can use the `LITEM$`, `LITEMEXISTS%`, `MAXLITEM`, and `COUNTLITEMS` commands to get information about list items. Use the `MITEM$`, `MITEMEXISTS%`, `MAXMITEM`, and `COUNTMITEMS` to get information about menu items.

You can construct and display another dialog inside the button callback.

Dialogs are not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

Time and Date

SCL supports time and date values. These values are simply numbers, expressed as the number of seconds elapsed since midnight, January 1st, 1901. You can use these numbers like any other numbers, adding, subtracting them, passing them as function parameters and storing them in numerical variables.

To get the current local time value, use the reserved variable `TIME`.

To calculate an arbitrary time value from month, day, year, hours, minutes, and seconds, use the `MKTIME` function.

To calculate a time difference, simply subtract two time values. Since the time values represent seconds since the beginning of the 20th century, this will give you an interval in seconds.

You can convert between local time and Greenwich mean time using the `GMT` and `LCTIME` functions.

You can write a time value to a string using the `TIME$` function. You can write a time interval (a number of seconds) to a string using the `INTVMINS$` and `INTVHRS$` functions.

There are also functions that allow you to extract the month, day, year, hours, minutes, seconds, or day of the week from a time value. These functions are listed in the *Function Reference* under *Time and date functions*.

File Input and Output and Network Connections

File Input and Output

You can access files on your file system from within SCL programs. When you open a file, you must assign a file number to it. Each file number must be a unique integer between 0 and 4,294,967,295. No two open files can have the same file number.

To open and close a file, use the `OPEN` and `CLOSE` commands.

Files can be opened as text files or binary files. If you open a file as a text file, SCL will translate CR+LF pairs to CRs when reading from the file, and vice-versa when writing to it.

Each file has a current file position maintained by the operating system. This is the place within the file from which the next read or write operation will begin. You can get and set the file position using the `FPOS` function and the `SETFPOS` command. Every time you read or write a certain number of bytes, the file position is advanced to the end of the string read or written.

You can use the `FLEN` function to determine the length of a file.

You can use the `LIMITFLEN` command to keep a file from growing continually until it fills all available disk space.

<p>Caution: The file position and file length are represented in bytes from the beginning of the file, not characters. If you opened the file as a text file, CR+LF pairs will</p>

be counted as two bytes, but only read as one character. This means that the file position and length in text files do not reflect the number of characters read, or the number of characters contained in the file.

To write and read to and from a file, use the `PRINT#` and `INPUT#` commands.

All open files are automatically closed when the program ends.

Network Connections

You can make TCP/IP network connections from within SCL programs. Network connections are treated the same as files: each network connection must have a file number, just like a file.

To open a connection with a TCP/IP server, use the `CONNECT` command. To disconnect, use the `CLOSE` command. Use the `PRINT#` and `INPUT#` commands to send and receive data.

SCL does not have any built-in knowledge about the protocols used by the server you are connecting to. You must implement the protocol yourself using the `PRINT#` and `INPUT#` commands. The `CONNECT` command is designed to communicate with servers that were specifically designed to communicate with SCL programs. Although it is theoretically possible to connect, say, to an FTP server using an SCL program, it is fairly complicated and requires a knowledge of the details of the FTP protocol.

You cannot use the `FPOS` function, the `SETFPOS` command, or the `LIMITFLEN` command with network connections, since network connections have neither a length nor a position.

File input and output and network connections are not available within programs for sources, checksums, and SABus response data.

Decoding and Encoding Data

You can define one or more data decoders and encoders for an SCL program in the New Program dialog. These decoders and encoders are used to construct or interpret SCL strings using special commands and functions.

A decoder tells the command or function how to parse a numerical, Boolean, or string value from a data string. An encoder tells the command or function how to write a numerical, Boolean, or string value to a data string.

Each decoder and each encoder has a decoder or encoder number, which is used to tell the function or command which decoder or encoder to use.

Decoding or Encoding a Single Value

You can use an encoder to encode a single value using the `ENCODE$` function. To decode a single value from a string, use the `DECODE`, `DECODE$`, or `DECODE%` functions. These functions look for a value at the beginning of the string, and they ignore any characters that appear after the value.

You can also decode a string using a decoding you create on the fly out of three regular expressions using the `DECODEREGEX$` function.

Decoding a Sequence of Values from a String

If a string contains more than one value, you can use the `PARSEDEC` and `PARSEREGEX` commands to parse the values. These functions take a numerical variable that holds the position of the value you wish to parse. After the command has decoded the value, the command will set the position variable to the first position *after* the value it found. This allows you to then parse the next value in the string using the same position variable.

The first character in a string is position 1. You do not need to set the position variable to 1 to start parsing at the beginning of the string, however. The parsing functions start parsing at the first character if the position is 0.

To start parsing a string, set the position variable to 0 or 1. Then, call `PARSEDEC` or `PARSEREGEX` repeatedly with the same position variable to parse all the values. Since a numerical variable is 0 when it is first used, you only need to set the position variable manually if you have used it before.

To parse three numbers from `data_string$` using decoders number 1, 2, and 3, use the following code:

```
position=0
PARSEDEC data_string$,1,parsed_number1,position
PARSEDEC data_string$,2,parsed_number2,position
PARSEDEC data_string$,3,parsed_number3,position
```

You can add a number to the position to skip a fixed number of data bytes. If you know that there are exactly 2 bytes between the numbers, you can use the following code:

```
PARSEDEC data_string$,1,parsed_number1,position
position=position+2
PARSEDEC data_string$,2,parsed_number2,position
position=position+2
PARSEDEC data_string$,3,parsed_number3,position
```

Similarly, you can skip the first 10 characters in a string by setting the position variable to 11 initially.

Use the `SKIPDEC`, and `SKIPREGEX` commands to skip extra data that appears between the values. If there must be a comma between the numbers, use the following code:

```
PARSEDEC data_string$,1,parsed_number1,position
SKIPREGEX data_string$,"", position
PARSEDEC data_string$, 2,parsed_number2,position
SKIPREGEX data_string$,"", position
PARSEDEC data_string$,3,parsed_number3,position
```

To check if the entire string has been parsed, check if `position=LEN(data_string$)`.

Encoding a Sequence of Values to a String

To write a series of values to the string, use the `APPENDSTR`, `APPENDCSTR`, `APPENDHEX`, and `APPENDENC` commands. Each of these commands append data to the content of a string variable.

The `APPENDSTR`, `APPENDCSTR`, and `APPENDHEX` commands to append a fixed string to the variable. The `APPENDENC` command uses an encoding to append a numerical, string, or Boolean value to the variable.

To construct a data string, start with a string variable that contains no data. Use the appending commands to add values to the string variable. To write three numbers to a string using encoders 1, 2, and 3, use the following code:

```
data_string$=""
APPENDENC data_string$,1,number1
APPENDENC data_string$,2,number2
APPENDENC data_string$,3,number3
```

To add commas between the numbers, use the following code:

```
APPENDENC data_string$,1,number1
APPENDSTR data_string$,","
APPENDENC data_string$,2,number2
APPENDSTR data_string$,","
APPENDENC data_string$,3,number3
```

You can specify more than one value for an append command. To encode three numbers, one after the other, all using encoding number one, use the following:

```
APPENDENC data_string$,1,number1,number2,number3
```

Serial Communication

Sending Commands

SCL allows you to send a command defined in a device driver to a serial port. Use the `SENDCMD` command to send the command. You must specify the port, the device, the command, and all of its parameters when you use `SENDCMD`.

U.P.M.A.C.S. will send the command to the port, and wait for a response, if required. If the command times out or the device returns an error, and the device driver specifies a number of retries for timeouts or for the error returned, `SENDCMD` will try to resend the command the specified number of times.

Use the `DRVSUCCESS%`, `DRVTIMEOUT%`, and `DRVERROR%` reserved variables to determine if the command was sent successfully. Use the `DRVDATA$` reserved variable or the `DRVNDATA$` function to access the data returned by the equipment. Use the `DRVERROR` and `DRVERROR$` reserved variables, or the `DRVNERROR`, and `DRVNERROR$` functions to access the error codes returned by the equipment.

Any registers that get their data from the command's response will be updated when you send a command.

Caution: SCL will send commands regardless of whether the device has been properly initialized or not. If it is important that the device has been initialized, use the `DRVREADY%` function to determine if the driver is ready.

If you want to send arbitrary data that is not part of a serial device command to a serial port, you can use the `SENDSTR` and `SENCBIN` commands. You can also use the `SENDSTR` and `SENCBIN` commands to send custom commands to devices that use legacy device drivers. Use the `SENDCREPLY` command to send pre-defined replies to a device that uses a legacy device driver.

Synchronizing Port Access

Since U.P.M.A.C.S. is a fully multitasking system, any number of SCL programs can run at the same time, while all ports are still being polled. It will usually be necessary to make sure that no other SCL programs can access the serial port or ports that a program needs, and that no polls are sent while the program is using the port.

For single commands, this synchronization is automatic.

If you need to send more than one command, and want to ensure that you are not interrupted, you can “grab” one or more ports using the `GRAB` command. This will give you exclusive access to the ports until the program ends, or you release them using the `RELEASE` command.

To avoid deadlock between SCL programs, you cannot grab additional ports when you already have exclusive access to other ports. You must release the ports you have grabbed before you can grab any additional ports. For the same reason you cannot access ports you have not grabbed, if you have already grabbed others.

If a program calls a child program using the `CALL` or `DRVCALL` commands, the child will have exclusive access to any ports the parent has grabbed. Child programs can only grab ports themselves if the parent program does not have any grabbed ports. Child programs cannot release ports the parent has grabbed.

Serial communication is not available within programs for sources, checksums, and SABus response data.

Programs for Sources, Checksums, and SABus Response Data

SCL programs are used in processor and summary serial data object and register sources to determine the value of the data object or register. Processor sources use a section extracted from a response, whereas summary sources use the values of other objects/registers. SCL programs are also used to do custom checksum calculations, and to specify the data of SABus processor response data objects.

You can access the tag of the data object or register that triggered the program using the `TRIGGER$` reserved variable. For programs for serial data object sources, you can use the `TRIGGERPRM`, `TRIGGERPRM$`, and `TRIGGERPRM%` functions to access the data object’s parameters. Use `TRIGGERPRM` for digital and analog parameters, `TRIGGERPRM$` for string parameters, and `TRIGGERPRM%` for bistate parameters.

Accessing the Data (Processor Sources and Checksums Only)

The response data that should be used for the value of the data object or register (processor sources), or the command or response data whose checksum should be calculated, can be accessed in two ways. The reserved string variable `BUFFER$` can be used to access the entire data as a string; the `BUFFER` function allows access to single data bytes as numbers.

Specifying the Data Object/Register Value or Checksum

You specify the resulting data, value, or checksum by setting a variable with a specific name. The result variable is a normal user variable and can be used like any other user variable. The names of the special variables are not reserved for use in programs for sources, checksums, and SABus response data. You can use variables with the same name in any type of program.

The names of the result variables for the different types of programs are listed below

- **Checksums**

For checksum programs, you must write the checksum to the variable **RESULT**.

- **Serial Data Objects and Registers**

For processor and summary sources, a special variable must be set to specify the value that the register should assume.

Object/Register type	Variable name	Variable type
bistate register	RESULT%	Boolean
digital register	RESULT	numerical
analog register	RESULT, GLRESULT	numerical
string register	RESULT\$	string

If you do not use the variable listed for the type, the object or register will go into its error state, (except for the GLRESULT variable, which does not need to be set). If you encounter an error parsing the data, do not use the result variable, so that an error will be flagged.

A bistate object or register will go into its on/alarm state if you set RESULT% to true, or into its off/alarm clear state if you set it to false. Registers with a response time will only assume the specified state after the response time has elapsed.

A digital object or register will assume the value specified in RESULT only if it is an integer between 0 and \$FFFFFFF. Otherwise, it will go into its error state.

An analog object or register will assume the value specified in RESULT. You can specify the greater / less status of the object or register using the GLRESULT numerical variable. Set GLRESULT to a number greater than 0 for "greater than", a number less than 0 for "less than", and to 0 for a normal value (equal). You do not need to use the GLRESULT variable. If you do not use it, it will be assumed to be 0.

If an analog object or register has a size of more than one value, RESULT and GLRESULT must be 1-dimensional arrays rather than regular user variables. The values with indices 1, 2, 3, etc. will take the values of RESULT[1], RESULT[2], RESULT[3], etc. and GLRESULT[1], GLRESULT[2], GLRESULT[3], etc.. The index of the first value is 1. If an analog object or register has a size of one value, you can either use regular variables, or use arrays and specify the value in RESULT[1] and GLRESULT[1].

A string object or register will also simply assume the value specified in RESULT\$.

You can also (auto-)mask the target object or register by setting the special variable **MASKRESULT%** to true. If you set MASKRESULT% to true, the other result variables will be ignored.

▪ SABus Response Data Objects

For SABus response data, you must write the data for the response section to the variable **RESULT\$**. If you do not set the variable, or if you set it to a value that contains non-printable characters (ASCII \$00-\$1F and \$7F-\$FF), the response data section will be empty.

Restrictions on Functions and Commands

Programs for sources and SABus response data, and all programs called by them using the `CALL` or `DRVCALL` command, are limited to 500 instructions, unless otherwise specified in the program's properties.

There are many commands and functions that you may not use in programs for sources, checksums, or SABus response data:

- User message commands
- Dialog commands
- File input and output and network connection commands and functions
- Serial communications commands
- Serial communications functions for accessing the response data or status
- Commands that modify registers or serial data objects or their values, or the `SETPARAM` command
- Logging commands
- RTS and SABus response commands
- The `STOPNET` and `STARTNET` commands
- The `CALLRMT`, `RUN`, `DRVRUN`, `RUNRMT`, and `LAUNCH` commands
- The `DELAY` command

Even though you cannot use register and data object *commands*, you can use register and data object *functions*. You can also use the `CALL` and `DRVCALL` commands.

Check the description for the individual commands to see whether a command can be used in programs for sources, checksums, or SABus response data.

Programs for SABus Commands

Programs are used in SABus commands to take any actions necessary as a consequence of the command, and to return a reply or error message to the remote system.

The variables you specified in variable command parameters will be set to the values of the parsed from the SABus command packet, rather than their default values. This allows you to access the parameter values.

To return a reply to the remote system, use the `SABUSREPLY` command. To send an error message, use the `SABUSERROR` command. The uplink port that received the command that triggered the program will be locked and no new SABus requests processed on it until you use either the `SABUSREPLY` or the `SABUSERROR` command. Using either command will release the uplink port, and further request from the remote system will be then

processed. You can only send one reply or error message to any command, a second `SABUSREPLY` or `SABUSERERROR` will generate an error. If you do not use either the `SABUSREPLY` or the `SABUSERERROR` command, the uplink port will be released when your program exits.

You should always use exactly one `SABUSREPLY` command or one `SABUSERERROR` command, as every SABus request should have exactly one reply. Although it is possible to end the program without having used either command, this is bad practice, because the request that triggered the program will then not generate a reply.

The uplink port error messages built into U.P.M.A.C.S. all use three capital letters as an error code followed by the parameters, if applicable. For consistency's sake, you should use the same format for your own error messages. Whenever applicable, you should send error codes of the built-in error messages. E. g., if parameter 4 has an illegal value, use the following code to send the error response:

```
SABUSERERROR "PRM04"
```

You should always send a reply or error message as early in the program as possible. This will ensure that the uplink port does not stay blocked for an unnecessary amount of time, and that there is no excessive delay between an SABus command and its response.

You can access the name of the SABus command that triggered the control using the `TRIGGER$` reserved variable.

Restrictions on Functions and Commands

There are some commands and functions that you may not use in programs for SABus commands:

- User message commands
- Dialog commands
- RTS commands

Check the description for the individual commands to see whether a command can be used in programs for sources, checksums, or SABus response data.

Device Driver Programs

You can define SCL programs inside device drivers. These programs can be used to calculate checksums, or for processor and summary sources of serial device data objects. Device driver programs can also be called be invoked programs using the `DRVCALL` and `DRVRUN` commands.

Some special considerations apply when accessing station objects from a device driver program:

- **Registers and legacy parameters**

Device driver programs cannot access registers or legacy parameters. This means that commands and functions that access registers or legacy parameters cannot be used in device driver programs.

- **Serial ports and devices**

Device driver programs can only access their own device driver and serial port. Whenever you have to specify a serial port or serial device in a command or function, you must use the empty string "" for the tag of the port or device. The right port and device will be used automatically.

- **SCL programs**

If you use the `CALL` or `RUN` command, a program from the same device driver will be invoked, rather than a normal program.

You can use the `DRVPRM`, `DRVPRM$`, and `DRVPRM%` functions to access the device driver parameters. Use `DRVPRM` for digital and analog parameters, `DRVPRM$` for string parameters, and `DRVPRM%` for bistate parameters.

Invoking SCL Programs From Within an SCL Program

You can call another SCL program from within an SCL program. This can be done in two ways.

You can invoke another program as an independent program using the `RUN` and `DRVRUN` command. The command does not wait until the invoked program finishes. You can schedule programs to be executed at a later time using the `RUN` or `RUNDRV` command.

You can also call a program as a child program using the `CALL` or `DRVCALL` command. The command does not return until the called program has finished, and the called program inherits certain properties from its parent, including message and dialog box titles, and grabbed serial ports.

Child programs can access variables of their parent programs using the `SETPVAR` command and the `PVAR`, `PVAR$`, and `PVAR%` functions.

`CALL` and `RUN` are used to run normal SCL programs defined outside a device driver. `DRVCALL` and `DRVRUN` are used to run device driver programs. If you use `CALL` or `RUN` within a device driver program, however, a program from the same driver will be executed, not a normal program.

For both independent and child programs, you can specify program arguments for the invoked program. The arguments consist of variable-value pairs.

Executing Programs On a Remote Computer

You can use the `CALLRMT` and `RUNRMT` to execute programs on a remote computer that is also running U.P.M.A.C.S.. The program must be a program defined in the station file of the remote computer, not the local computer.

The remote program will run on the remote computer, and use the registers, serial port, log files, etc. of the remote computer. Only user messages and dialogs will be shown on the local computer.

You can only execute programs if the remote computer has insecure remote control enabled. See *Network Security* in the U.P.M.A.C.S. *Operator's Manual* for details.

The `CALLRMT`, `RUN`, `DRVRUN`, and `RUNRMT` commands are not available within programs for sources, checksums, and SABus response data.

RTS Controls

RTS controls are specialized SCL programs that are executed by another application via the network. The application provides a number of string parameters, called RTS parameters. The control can access the parameters using the `RTSPRM$` function.

The RTS control can send two kinds of replies to the application that triggered it:

- To send an information message, use the `RTSEND` command.
- To send an error message, use the `RTSEERROR` command.

RTS controls do not need to be specially marked, nor do they have a special format. If RTS controls are enabled in the network security setting of the U.P.M.A.C.S. Operate System, the remote application can execute any SCL program as an RTS control.

For information on how to write applications that use the RTS protocol, contact UPMACS Communications, inc.

Restrictions on Functions and Commands

There are some commands and functions that you may not use in RTS controls:

- User message commands
- Dialog commands
- SABus commands

Check the description for the individual commands to see whether a command can be used in programs for sources, checksums, or SABus response data.

RESERVED VARIABLE REFERENCE

Serial Communication Reserved Variables

■ The **DRVSUCCESS%** Reserved Variable

Determines whether a valid response was received to the last command sent.

Syntax:

DRVSUCCESS%

DRVSUCCESS% is true if a valid response was received to the last command sent using the **SENDCMD** command. **DRVSUCCESS%** is always true for commands that do not expect a response, or if you sent data to the serial port using a command other than **SENDCMD**.

If **DRVSUCCESS%** is true, use the **DRVDATA\$** reserved variable or the **DRVNDATA\$** function to retrieve the response data.

If **DRVSUCCESS%** is false, use the **DRVTIMEOUT%** and **DRVERROR%** reserved variables to determine what went wrong.

Note: **DRVSUCCESS%** can also be used to determine if a valid response was returned to a custom command sent to a device that uses a legacy device driver using **SENDSTR** or **SENBIN**.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

■ The **DRVTIMEOUT%** Reserved Variable

Determines whether the last command sent timed out.

Syntax:

DRVTIMEOUT%

DRVTIMEOUT% is true if a timeout occurred waiting for the response to the last command sent using the **SENDCMD** command. **DRVTIMEOUT%** is always false for commands that do not expect a response, or if you sent data to the serial port using a command other than **SENDCMD**.

Note: **DRVSUCCESS%** can also be used to determine if a custom command sent to a device that uses a legacy device driver using **SENDSTR** or **SENBIN** timed out.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

■ The **DRVERROR%** Reserved Variable

Determines whether an error response was received to the last command sent.

Syntax:

DRVERROR%

DRVSUCCESS% is true if an error response was received to the last command sent using the SENDCMD command. DRVERROR% is always false for commands that do not expect a response, or for commands that do not have an error response defined. DRVERROR% is also false if you sent data to the serial port using a command other than SENDCMD.

If DRVERROR% is true, use the DRVERROR and DRVERROR\$ reserved variables and the DRVNERROR and DRVNERROR\$ functions to retrieve the error codes.

Note: DRVSUCCESS% can also be used to determine if an error response was returned to a custom command sent to a device that uses a legacy device driver using SENDSTR or SENDBIN.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

■ The **DRVDATA\$** Reserved Variable

Contains the data of the first response data element of the last command sent.

Syntax:

DRVDATA\$

DRVDATA\$ contains the content of the first response data element of the response to the last command sent using the SENDCMD command. DRVDATA\$ is an empty string if the command has no response, if its response has no response data response elements, or if no valid response was received. Use the DRVSUCCESS% reserved variable to determine if a valid response was received.

DRVDATA\$ is also an empty string if you sent data to the serial port using a command other than SENDCMD.

Use the DRVNDATA\$ function to retrieve the data of response data elements other than the first one.

Note: DRVDATA\$ can also be used to retrieve the data (without the prefix and suffix) of the response to a command sent to a device that uses a legacy device driver using SENDCMD, SENDSTR, or SENDBIN.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

■ The **DRVERROR** Reserved Variable

Contains the main error code the device returned to the last command sent.

Syntax:**DRVERROR**

DRVERROR contains the main error code returned by the device to the last command sent using the SENDCMD command. DRVERROR is 0 if a valid response was received or the command timed out, if the command has no main error code, or if the main error code is a string error code. Use the DRVERROR% reserved variable to determine if an error response was received.

DRVERROR is also 0 if you sent data to the serial port using a command other than SENDCMD.

Use the DRVERROR\$ reserved variable to retrieve the main error code if it is a string. Use the DRVNERROR function to retrieve numerical error codes other than the main error code.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

■ The DRVERROR\$ Reserved Variable

Contains the main error code the device returned to the last command sent.

Syntax:**DRVERROR\$**

DRVERROR\$ contains the main error code returned by the device to the last command sent using the SENDCMD command. DRVERROR\$ is an empty string if a valid response was received or the command timed out, if the command has no main error code, or if the main error code is a numerical error code. Use the DRVERROR% reserved variable to determine if an error response was received. DRVERROR\$ is also an empty string if you sent data to the serial port using a command other than SENDCMD.

Use the DRVERROR reserved variable to retrieve the main error code if it is numerical. Use the DRVNERROR\$ function to retrieve string error codes other than the main error code.

Note: DRVERROR\$ can also be used to retrieve the error response to a command sent to a device that uses a legacy device driver using SENDCMD, SENDSTR, or SENDBIN.

This reserved variable is not available within programs for sources, checksums, and SABus response data.

Miscellaneous Reserved Variables

■ The PRGNAME\$ Reserved Variable

Contains the name of the SCL program being executed.

Syntax:**PRGNAME\$**

PRGNAME\$ contains the name (not the tag) of the SCL program that is currently being executed. If the program is a child program executed using the CALL, DRVCALL, or CALLRMT command, the name of the parent program is used.

■ The **TIME** Reserved Variable

Contains the current time.

Syntax:

TIME

TIME contains the current time expressed as the number of seconds elapsed since midnight, January 1st, 1901.

■ The **USR\$** Reserved Variable

Contains the name of the current user.

Syntax:

USR\$

USR\$ contains the name of the user who is currently logged on to the local station. Use USRLVL to determine the current user's clearance level, and USRPRV% to determine if the current user has privileges to perform a certain action.

■ The **USRLVL** Reserved Variable

Contains the clearance of the current user.

Syntax:

USRLVL

USRLVL contains the clearance of the user who is currently logged on to the local station. The levels have the following meaning:

Value	Clearance
0	there is no one signed on, or the current user has no clearance
1	the current user is an operator
2	the current user is a supervisor
3	the current user is an administrator

Use USRPRV% to determine if the current user has privileges to perform a certain action.

■ The **NETUP%** Reserved Variable

Determines whether remote connections are enabled.

Syntax:

NETUP%

NETUP% is false if remote connections have been disabled using the STOPNET command. Use STARTNET to re-enable remote connections. If NETUP% is false, no remote connections can be initiated from other computers, including connections to the local station, insecure remote control connections, and network register source connections.

NETUP% only checks whether networking has been disabled using the STOPNET command, it does not check whether remote connections are enabled in the network security settings. If remote connections have not been disabled using STOPNET, NETUP% will be true even if networking has been disabled in the network security settings.

Special Purpose Reserved Variables

■ The BUFFER\$ Reserved Variable

Contains the data in the data buffer.

Syntax:
BUFFER\$

BUFFER\$ contains the data that is to be evaluated. Use the BUFFER function to retrieve single bytes from within the buffer.

This reserved variable is only available within programs for processor sources or checksums.

■ The TRIGGER\$ Reserved Variable

Contains the tag of the object that triggered this program.

Syntax:
TRIGGER\$

TRIGGER\$ contains the tag of the following object:

Program type	Object
Automatic control	Register
Device control	Serial port
Serial data object source	Data object
Register source	Register
SABus command	Command object

To retrieve the tag of a device for a device of a device control, use the TRIGGERDRV\$ reserved variable. To retrieve the parameters of the serial data object for programs for data object sources, use the TRIGGERPRM, TRIGGERPRM\$, and TRIGGERPRM% reserved variables.

Note: For message controls of devices that use legacy device drivers, you can use the `TRIGGERMSG$` reserved variable to retrieve the tag of the message that triggered the control.

This reserved variable is only available within automatic and device controls, programs for processor and summary sources, and programs for SABus commands.

■ The `TRIGGERDRV$` Reserved Variable

Contains the tag of the device that triggered this device control.

Syntax:

TRIGGERDRV\$

`TRIGGERDRV$` contains the tag of the device that triggered a device control. To retrieve the tag of the device's serial port, use the `TRIGGER$` reserved variable.

Note: For message controls of devices that use legacy device drivers, you can use the `TRIGGERMSG$` reserved variable to retrieve the tag of the message that triggered the control.

This reserved variable is only available within device controls.

Legacy Object Reserved Variables

■ The `TRIGGERMSG$` Reserved Variable

Contains the tag of the legacy device driver message that triggered this device control.

Syntax:

TRIGGERMSG\$

`TRIGGERMSG$` contains the tag of the message that triggered a device control. To retrieve the tag of the device use the `TRIGGERDRV$` reserved variable. To retrieve the tag of the device's serial port, use the `TRIGGER$` reserved variable.

This reserved variable is only available within a legacy device's message control.

FUNCTION REFERENCE

Mathematical Functions

■ The **ABS** Function

Calculates the absolute value of a number.

Syntax:

ABS(n)

Parameters:

n: the number of which to calculate the absolute value

■ The **SQRT** Function

Calculates the square root of a number.

Syntax:

SQRT(n)

Parameters:

n: the number

If n is negative, **SQRT** generates an error.

■ The **SIN** Function

Calculates the sine of an angle.

Syntax:

SIN(angle)

Parameters:

angle: the angle, in radians

■ The **COS** Function

Calculates the cosine of an angle.

Syntax:

COS(angle)

Parameters:

angle: the angle, in radians

■ The **TAN** Function

Calculates the tangent of an angle.

Syntax:

TAN(angle)

Parameters:

angle: the angle, in radians

If angle is an odd multiple of $\pi/2$ ($\pi/2$, $3\pi/2$, $5\pi/2$, $-\pi/2$, $-3\pi/2$, etc.), **TAN** generates an error.

■ The **EXP** Function

Calculates a power of e.

Syntax:

EXP(exponent)

Parameters:

exponent: the exponent to which to raise e

■ The **LN** Function

Calculates the natural logarithm of a number.

Syntax:

LN(n)

Parameters:

n: the number

If n is negative or zero, **LN** generates an error.

■ The **LOG2** Function

Calculates the logarithm base 2 of a number.

Syntax:

LOG2(n)

Parameters:

n: the number

If n is negative or zero, **LOG2** generates an error.

■ The **LOG10** Function

Calculates the logarithm base 10 of a number.

Syntax:**LOG10**(n)Parameters:

n: the number

If n is negative or zero, LOG10 generates an error.

■ The MOD Function

Calculates the modulus of a number.

Syntax:**MOD**(n, order)Parameters:

n: the number

order: the order of the modulus (the divisor)

The sign of the modulus is always the same as that of order.

If order is zero, MOD generates an error.

*Note for C Programmers*Definition of the modulus

The modulus of order **o** of a number **n** is a number **m** such that there exists an integer **i** where $n = o \cdot i + m$ and $|m| < |o|$.

Note: Technically, the modulus operator is only defined for integral n and positive integral orders. Since extending the definition of a modulus to real arguments poses no difficulties as long as the order is non-zero, the MOD function allows real arguments, and negative orders.

Note for C Programmers: The ANSI C library function *fmod* does not calculate the modulus of a number, but the remainder of a division. The sign of the remainder is defined to be the same as that of the number, whereas the sign of the modulus is defined to be the same as that of the divisor. The remainder and modulus will therefore differ if the sign of the number is not the same as that of the divisor:

number	divisor	modulus	remainder
10	3	1	1
-10	3	2	-1
10	-3	-2	1
-10	-3	-1	-1

Those familiar with LISP will know the difference between the *mod* and *rem* functions in that language. The SCL MOD function behaves the same as the LISP *mod* function.

Examples:

```
MOD(10,4)      = 2
MOD(2.5,0.75) = 0.25
MOD( 5, 1.5)   = 0.5
MOD(-5, 1.5)   = 0.5
MOD( 5,-1.5)   = -0.5
MOD(-5,-1.5)   = -0.5
```

■ The RND Function

Rounds a number.

Syntax:

RND(n)

Parameters:

n: the number to round

| RND(n) - n | is always less than or equal to 0.5, i.e. RND(n) is never more than 0.5 away from n.

■ The RNDDWN Function

Rounds a number down.

Syntax:

RNDDWN(n)

Parameters:

n: the number to round

RNDDWN(n) is always less than or equal to n.

■ The RNDUP Function

Rounds a number up.

Syntax:

RNDUP(n)

Parameters:

n: the number to round

RNDUP(n) is always greater than or equal to n.

String Manipulation Functions

■ The **LEN** Function

Determines the number of characters in a string.

Syntax:

LEN(string\$)

Parameters:

string\$: the string

If string\$ is empty, **LEN**(string\$) is 0.

■ The **LEFT\$** Function

Extracts the leftmost characters of a string.

Syntax:

LEFT\$(string\$,n)

Parameters:

string\$: the source string

n: the number of characters to extract

If n is greater than the length of string\$, **LEFT\$** returns the entire string.

If n is negative or not an integer, **LEFT\$** reports an error.

■ The **RIGHT\$** Function

Extracts the rightmost characters of a string.

Syntax:

RIGHT\$(string\$,n)

Parameters:

string\$: the source string

n: the number of characters to extract

If n is greater than the length of string\$, **RIGHT\$** returns the entire string.

If n is negative or not an integer, **RIGHT\$** reports an error.

■ The **MID\$** Function

Extracts an arbitrary substring from a string.

Syntax:

MID\$(string\$,position,n)

Parameters:

`string$`: the source string
`position`: the position of the substring within the string
`n`: the number of characters to extract

A position of 1 indicates that the substring begins with the first character of `string$`.

If `position` is greater than the length of `string$`, `MID$` returns an empty string.

If there are less than `n` characters in `string$` after `position`, then `MID$` returns all characters in the string from `position` on.

If `position` is less than one or not an integer, `MID$` generates an error.

If `n` is negative or not an integer, `MID$` generates an error.

■ The `POS` Function

Determines the position of a substring within a string.

Syntax:

POS(`string$`, `sub_string$`)

Parameters:

`string$`: the string to search
`sub_string$`: the string to search for

`POS` returns the position of the first occurrence of `sub_string$` in `string$`. A return value of 1 indicates that `sub_string$` is located at the beginning of `string$`. If `sub_string$` is not contained in `string$`, `POS` returns -1.

■ The `REGEXPOS` Function

Determines the position of a regular expression within a string.

Syntax:

REGEXPOS(`string$`, `reg_ex$`)

Parameters:

`string$`: the string to search
`reg_ex$`: the regular expression to search for

`REGEXPOS` returns the position of the first occurrence of `reg_ex$` in `string$`. A return value of 1 indicates that `reg_ex$` is located at the beginning of `string$`. If `reg_ex$` is not contained in `string$`, `REGEXPOS` returns -1.

See *Appendix A: Regular Expressions* in the *Developer's Manual* for details on regular expressions.

■ The `REGXEND` Function

Determines the position of the end of regular expression within a string.

Syntax:**REGEXEND**(string\$,reg_ex\$)Parameters:

string\$: the string to search

reg_ex\$: the regular expression to search for

POS returns the position of the first character *after* the first occurrence of reg_ex\$ in string\$. A return value of 3, e.g., indicates that reg_ex\$ matches the first two characters at the beginning of string\$. If reg_ex\$ is not contained in string\$, REGEXEND returns -1.

See *Appendix A: Regular Expressions* in the *Developer's Manual* for details on regular expressions.

String Conversion Functions

■ The CHR\$ Function

Makes a string consisting of a character with a specific character code.

Syntax:**CHR\$**(code)Parameters:

code: the one-byte character code

If code is less than 0, CHR\$ creates a character whose character code is code +256. This allows you to use CHR\$ to create binary strings containing signed byte values.

If code is not an integer between -128 and 255, CHR\$ generates an error.

■ The ASCII Function

Determines the character code of the first character in a string.

Syntax:**ASCII**(character\$)Parameters:

character\$: the string containing the character

Despite its name, the ASCII function works for characters with codes greater than 127.

Use ASCII to extract the first byte of a string containing binary data. ASCII always returns a value between 0 and 255. To extract a signed byte value (between -128 and 127) use the SASII function.

ASCII return 0 if character\$ is an empty string.

■ The **SASCII** Function

Determines the signed byte equivalent of the character code of the first character in a string.

Syntax:

SASCII(character\$)

Parameters:

character\$: the string containing the character

The **SASCII** function returns negative numbers for characters with codes greater than 127. The value returned is the character code minus 256.

Use **SASCII** to extract the first byte of a string containing binary data. **SASCII** always returns a value between -128 and 127. To extract an unsigned byte value (between 0 and 255) use the **ASCII** function.

SASCII return 0 if character\$ is an empty string.

■ The **STR\$** Function

Writes out a string representation of a number using the decimal marker specified in the *Regional Settings* control panel.

Syntax:

STR\$(n)

Parameters:

n: the number to write out

STR\$ creates a string using the current language settings in the *Regional Settings* control panel. Use this function if you want to display a number to the user or write it to the log. If you want to use the result string internally only, i.e. if it is intended for a parameter or another SCL program, or if you want to use it in an RTS response, use **ISTR\$**, **FMT\$**, or **ENCODE\$** instead.

■ The **ISTR\$** Function

Writes out a string representation of a number, always using a period as the decimal marker.

Syntax:

ISTR\$(n)

Parameters:

n: the number to write out

ISTR\$ creates a string using a period as the decimal marker, regardless of the current language settings in the *Regional Settings* control panel. Use this function if you want to use the result string internally only, i.e. if it is intended for a parameter or another SCL program, or if you want to use it in an RTS response. If you want to display a number to the user or write it to the log, use **STR\$** instead.

If you need more control over the exact format, use `FMT$`, `HEXFMT$`, `HEXFMT2$`, `BINFMT$`, `OCTFMT$`, or `ENCODE$`.

■ The `VAL` Function

Extracts a number written out in a string using the decimal marker specified in the *Regional Settings* control panel.

Syntax:

`VAL(string$)`

Parameters:

`string$`: the string that contains the number

`VAL` ignores any characters in the string that appear after the number.

If `string$` does not contain a number, or if there are characters other than spaces before the number, `VAL` returns 0.

`VAL` recognizes the base prefixes described in *Literal Values*. Exponents and decimals are ignored for non-decimal numbers.

`VAL` interprets a string that is formatted according to the current language settings in the *Regional Settings* control panel. Use this function if you want to parse a string input by the user. If you want to parse a string created using the `ISTR$` function or a string received in an equipment response or RTS parameter, use `IIVAL` or `DECODE`.

■ The `IIVAL` Function

Extracts a number written out in a string using a period as the decimal marker.

Syntax:

`IIVAL(string$)`

Parameters:

`string$`: the string that contains the number

`IIVAL` ignores any characters in the string that appear after the number.

If `string$` does not contain a number, or if there are characters other than spaces before the number, `IIVAL` returns 0.

`IIVAL` recognizes the base prefixes described in *Literal Values*. Exponents and decimals are ignored for non-decimal numbers.

`IIVAL` interprets a string that is formatted using a period as the decimal marker, regardless of the current language settings in the *Regional Settings* control panel. Use this function if you want to parse a string created using the `ISTR$` function or a string received in an equipment response or RTS parameter. If you want to parse a string input by the user, use `VAL` instead.

If you need more control over the exact format of the number, use `HEXIVAL`, `BINVAL`, `OCTIVAL`, or `DECODE`.

■ The **HEXVAL** Function

Extracts a hexadecimal number written out in a string.

Syntax:

HEXVAL(string\$)

Parameters:

string\$: the string that contains the number

HEXVAL ignores any characters in the string that appear after the number.

If *string\$* does not contain a hexadecimal number, or if there are characters other than spaces before the number, **HEXVAL** returns 0.

HEXVAL recognizes both capital (A-F) and small letters (a-f) as hexadecimal digits. Decimal points and exponents are ignored by **HEXVAL**.

If you need more control over the exact format of the number, use **DECODE**.

■ The **BINVAL** Function

Extracts a readable binary number (a series of the characters "1" and "0") written out in a string.

Syntax:

BINVAL(string\$)

Parameters:

string\$: the string that contains the number

BINVAL ignores any characters in the string that appear after the number.

If *string\$* does not contain a binary number, or if there are characters other than spaces before the number, **BINVAL** returns 0.

Decimal points and exponents are not recognized by **BINVAL**.

If you need more control over the exact format of the number, use **DECODE**.

■ The **OCTVAL** Function

Extracts an octal number written out in a string.

Syntax:

OCTVAL(string\$)

Parameters:

string\$: the string that contains the number

OCTVAL ignores any characters in the string that appear after the number.

If *string\$* does not contain an octal number, or if there are characters other than spaces before the number, **OCTVAL** returns 0.

Decimal points and exponents are ignored by **OCTVAL**.

If you need more control over the exact format of the number, use `DECODE`.

■ The `BCDVAL` Function

Extracts a binary coded decimal value from a string.

Syntax:

`BCDVAL(string$)`

Parameters:

`string$`: the string that contains the number

BCD stands for binary coded decimal. In binary coded decimal, each nibble (hex digit) in a byte represents one decimal digit.

A string containing the following four byte values:

\$20 \$34 \$00 \$50

represents the number 20340050.

`BCDVAL` uses all characters in the string. If `string$` is an empty string, `BCDVAL` returns 0.

If you need more control over the exact format of the number, use `DECODE`.

■ The `LOHIVAL` Function

Extracts an unsigned multi-byte value in low, high byte ordering (most significant byte last) from a string.

Syntax:

`LOHIVAL(string$)`

Parameters:

`string$`: the string that contains the number

`LOHIVAL` always returns an unsigned value. To extract a signed value, use the `SLOHIVAL` function.

`LOHIVAL` uses all characters in the string. If `string$` is an empty string, `LOHIVAL` returns 0.

If you need more control over the exact format of the number, use `DECODE`.

■ The `SLOHIVAL` Function

Extracts a signed multi-byte value in low, high byte ordering (most significant byte last) from a string.

Syntax:

`SLOHIVAL(string$)`

Parameters:

`string$`: the string that contains the number

`SLOHIVAL` returns a negative value if the most significant bit is 1. To extract an unsigned value, use the `LOHIVAL` function.

`SLOHIVAL` uses all characters in the string. If `string$` is an empty string, `SLOHIVAL` returns 0.

If you need more control over the exact format of the number, use `DECODE`.

■ The `HILOVAL` Function

Extracts an unsigned multi-byte value in high, low byte ordering (least significant byte last) from a string.

Syntax:

`HILOVAL`(`string$`)

Parameters:

`string$`: the string that contains the number

`HILOVAL` always returns an unsigned value. To extract a signed value, use the `SHILOVAL` function.

`HILOVAL` uses all characters in the string. If `string$` is an empty string, `HILOVAL` returns 0.

If you need more control over the exact format of the number, use `DECODE`.

■ The `SHILOVAL` Function

Extracts a signed multi-byte value in high, low byte ordering (least significant byte last) from a string.

Syntax:

`SHILOVAL`(`string$`)

Parameters:

`string$`: the string that contains the number

`SHILOVAL` returns a negative value if the most significant bit is 1. To extract an unsigned value, use the `HILOVAL` function.

`SHILOVAL` uses all characters in the string. If `string$` is an empty string, `SHILOVAL` returns 0.

If you need more control over the exact format of the number, use `DECODE$`.

■ The `FMT$` Function

Writes out a string representation of a number with a specified number of digits on either side of the decimal marker. `FMT$` always uses a period as the decimal marker.

Syntax:

`FMT$`(`n`, `left_digits`, `right_digits`)

Parameters:

`n`: the number to write out
`left_digits`: the number of digits to the left of the decimal point
`right_digits`: the number of digits to the right of the decimal point

If `right_digits` is 0, then `FMT$` will not include a decimal point in the output.

If `n` is too large to be represented in the required number of digits, excess digits are removed.

Examples:			
<code>n</code>	<code>left_digits</code>	<code>right_digits</code>	result
10.458	3	2	010.46
10.458	3	0	010
10.458	1	4	0.4580
10.458	0	2	.46
-10.458	3	2	-010.46
-10.458	3	0	-010
-10.458	1	4	-0.4580
-10.458	0	2	-.46

`FMT$` creates a string using a period as the decimal marker, regardless of the current language settings in the *Regional Settings* control panel. Use this function if you want to use the result string internally only, i.e. if it is intended for a parameter or another SCL program, or if you want to use it in an RTS response. If you want to display a number to the user or write it to the log, use `STR$` instead.

If `left_digits` or `right_digits` is smaller than 0 or greater than 100, `FMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `HEXFMT$` Function

Writes out a hexadecimal representation of a number with a specified number of digits, using capitals ('A'-'F').

Syntax:

`HEXFMT$(n,digits)`

Parameters:

`n`: the number to write out
`digits`: the number of digits

If `n` is too large to be represented in the required number of digits, excess digits are removed from the top.

`HEXFMT$` uses capital 'A'-'F' for digits. To format a number using the small letters 'a'-'f', use the `HEXFMT2$` function.

If `digits` is less than 1, `HEXFMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `HEXFMT2$` Function

Writes out a hexadecimal representation of a number with a specified number of digits, using small letters ('a'-'f').

Syntax:

`HEXFMT2$(n,digits)`

Parameters:

`n`: the number to write out

`digits`: the number of digits

If `n` is too large to be represented in the required number of digits, excess digits are removed from the top.

`HEXFMT2$` uses capital 'a'-'f' for digits. To format a number using the small letters 'A'-'F', use the `HEXFMT$` function.

If `digits` is less than 1, `HEXFMT2$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `BINFMT$` Function

Writes out a readable binary representation of a number (a series of the characters "1" and "0") with a specified number of digits.

Syntax:

`BINFMT$(n,digits)`

Parameters:

`n`: the number to write out

`digits`: the number of digits

If `n` is too large to be represented in the required number of digits, excess digits are removed from the top.

If `digits` is less than 1, `BINFMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `OCTFMT$` Function

Writes out an octal representation of a number with a specified number of digits.

Syntax:

`OCTFMT$(n,digits)`

Parameters:

`n`: the number to write out

`digits`: the number of digits

If `n` is too large to be represented in the required number of digits, excess digits are removed from the top.

If `digits` is less than 1, `OCTFMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `BCDFMT$` Function

Encodes a number into binary coded decimal value.

Syntax:

`BCDFMT$(n, bytes)`

Parameters:

`n`: the number to write out

`bytes`: the number of bytes to use for the result

BCD stands for binary coded decimal. In binary coded decimal, each nibble (hex digit) in a byte represents one decimal digit.

A string containing the following four byte values:

\$20 \$34 \$00 \$50

represents the number 20340050.

If `n` is too large to be represented in the required number of bytes, excess bytes are removed from the top.

If `n` is negative, or if `bytes` is less than 1, `BCDFMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The `LOHIFMT$` Function

Encodes a number into an unsigned multi-byte value in low, high byte ordering (most significant byte last).

Syntax:

`LOHIFMT$(n, bytes)`

Parameters:

`n`: the number to write out

`bytes`: the number of bytes to use for the result

If `n` is too large to be represented in the required number of bytes, excess bytes are removed from the top.

If `n` is negative, or if `bytes` is less than 1, `LOHIFMT$` generates an error.

If you need more control over the exact format of the number, use `ENCODE$`.

■ The **HILOFMT\$** Function

Encodes a number into an unsigned multi-byte value in high, low byte ordering (least significant byte last).

Syntax:

HILOFMT\$(n, bytes)

Parameters:

n: the number to write out

bytes: the number of bytes to use for the result

If **n** is too large to be represented in the required number of bytes, excess bytes are removed from the top.

If **n** is negative, or if **bytes** is less than 1, **HILOFMT\$** generates an error.

If you need more control over the exact format of the number, use **ENCODE\$**.

■ The **CCNV\$** Function

Creates a string containing non-printable characters from a string that has those characters encoded in special backslash sequences similar to those C compilers use.

Syntax:

CCNV\$(C_string\$)

Parameters:

C_string\$: the C style string

CCNV\$ enables you to easily specify strings containing non-printable characters (ASCII 00-1F and 7F-FF). To specify a non-printable character, use any of the following sequences of characters:

Sequence	Character	Code (hexadecimal)
\0	null character	\$00
\b	backspace	\$08
\t	tab	\$09
\n	linefeed	\$0A
\v	vertical tab	\$0B
\f	form feed	\$0C
\r	carriage return	\$0D

to specify any other non-printable character, use **\x** followed by two hexadecimal digits specifying the character code. Here are some examples:

Sequence	Character	Code (hexadecimal)
\x02	start transmission	\$02

<code>\x03</code>	end of transmis- sion	<code>\$03</code>
<code>\xFF</code>	delete	<code>\$FF</code>
<code>\xB7</code>	<code>\$B7</code>	
<code>\x69</code>	capital letter "E"	<code>\$69</code>

To specify a backslash, use two backslashes in a row:

Sequence	Character	Code (hexadecimal)
<code>\\</code>	backslash	<code>\$92</code>

Printable characters, with the exception of the backslash and double quotes, can just be entered plainly. If you feel so inclined, however, you can use a backslash followed by that character.

Here are some examples:

Sequence	Character	Code (hexadecimal)
<code>\a</code>	letter "a"	<code>\$97</code>
<code>\6</code>	digit six	<code>\$56</code>
<code>\/</code>	slash	<code>\$47</code>
<code>\R</code>	capital letter "R"	<code>\$82</code>

Note: `CCNV$` would theoretically convert `\` to the double quote character if it encountered it in a string. However, since the SCL interpreter will not allow double quotes in string literals, you *cannot* use the `\` sequence to specify double quotes. To generate the string:

```
Hi! My name is Fred "Barbarossa" Staufer!
```

You cannot use the following function call:

```
CCNV$("Hi! My name is Fred \"Barbarossa\" Staufer!") ← error!
```

You can use the `\x` character sequence and specify the ASCII code for the double quotes character instead:

```
CCNV$("Hi! My name is Fred \x34Barbarossa\x34 Staufer!")
```

You can also use the `QUT$` constant instead of `CCNV$`:

```
"Hi! My name is Fred "+QUT$+"Barbarossa"+QUT$+" Staufer!"
```

If `C_string$` ends in a backslash, or if it contains a `\x` that is not followed by two hexadecimal digits, `CCNV$` generates an error.

■ The **HCONV\$** Function

Creates a string containing characters with arbitrary character codes from a string in which these codes are written out in hexadecimal.

Syntax:

HCONV\$(hex_values\$)

Parameters:

hex_values\$: the string in which the hex values are written out

hex_values\$ has to be a string consisting of two digit hexadecimal values separated by single spaces. **HCONV\$** would convert the string:

```
"4F 7A 6F 6E 65 21"
```

to the following string:

```
"Ozone!"
```

If hex_values\$ does not conform to the format described above, or if there are any characters before the first or after the last hex value (including spaces), **HCONV\$** generates an error.

Data Decoding/Encoding Functions

■ The **DECODE** Function

Decodes a number from a string using a data decoder.

Syntax:

DECODE(string\$,decoder)

Parameters:

string\$: the data string

decoder: the number of the decoder to use

DECODE ignores any characters in string\$ that appear after the number. If string\$ does not contain a number in the specified encoding, or if there are characters before it, **DECODE** returns 0.

To decode more than one value from a data string, use **PARSEDEC**.

If decoder is not the number of a numerical decoder, **DECODE** generates an error.

■ The **DECODE\$** Function

Decodes a string from a string using a data decoder.

Syntax:**DECODE\$**(string\$,decoder)Parameters:

string\$: the data string

decoder: the number of the decoder to use

DECODE\$ ignores any characters in *string\$* that appear after the encoded string. If *string\$* does not contain a string in the specified encoding, or if there are characters before it, DECODE\$ returns an empty string.

To generate a decoder on the fly rather than using a pre-defined one, use the DECODEREGEX\$ function.

To decode more than one value from a data string, use the PARSEDEC and related commands.

If *decoder* is not the number of a string decoder, DECODE\$ generates an error.

■ The DECODE% Function

Decodes a Boolean value from a string using a data decoder.

Syntax:**DECODE%**(string\$,decoder)Parameters:

string\$: the data string

decoder: the number of the decoder to use

DECODE% ignores any characters in *string\$* that appear after the value. If *string\$* does not contain a value in the specified encoding, or if there are characters before it, DECODE% returns false.

To decode more than one value from a data string, use PARSEDEC.

If *decoder* is not the number of a Boolean decoder, DECODE% generates an error.

■ The DECODEREGEX\$ Function

Decodes a string from a string using three regular expressions.

Syntax:**DECODEREGEX\$**(string\$,prefix\$,pattern\$,suffix\$)Parameters:

string\$: the data string

prefix\$: the prefix pattern, or "" for none

pattern\$: the data pattern

suffix\$: the suffix pattern, or "" for none

The prefix pattern, if any, describes any data that appears in *string\$* before the encoded string. The data pattern describes the encoded string itself, and the suffix pattern,

if any, describes data that must appear after the encoded string (e.g. a terminating character).

`DECODEREGEX$` ignores any characters in `string$` that appear after the suffix (or data if there is no suffix). If `string$` does not contain a match for the specified expressions, or if there are characters before the prefix (or data if there is no prefix), `DECODEREGEX$` returns an empty string.

See *Appendix A: Regular Expressions* in the *Developer's Manual* for details on regular expressions.

To use a pre-defined encoding for parsing a string, use the `DECODE$` function.

To decode more than one value from a data string, use the `PARSEREGEX` and related commands.

If `pattern$` is not a valid regular expression, `DECODEREGEXP$` generates an error.

If `prefix$` or `suffix$` is not an empty string or a valid regular expression, `DECODEREGEXP$` generates an error.

■ The `ENCODE$` Function

Encodes a number, string, or Boolean value to a string using a data encoder.

Syntax:

`ENCODE$(encoder, number)`

or

`ENCODE$(encoder, string$)`

or

`ENCODE$(encoder, Boolean%)`

Parameters:

`encoder`: the number of the encoder to use

`number`: the number to encode

`string$`: the string to encode

`Boolean%`: the Boolean value to encode

`ENCODE$` returns a string that contains the value in the specified encoding.

To encode more than one value to a string, use the `APPENDENC` and related commands.

If `encoder` is not the number of an encoder for the right type of value, `ENCODE$` generates an error.

Checksum Functions

■ The `CHKSUM` Function

Calculates the simple checksum for a string.

Syntax:

CHKSUM(string\$)

Parameters:

string\$: the string whose checksum should be calculated

CHKSUM adds the character codes of all the characters in string\$.

■ The **CHKSUMLOHI** Function

Calculates the simple checksum for a string that contains multibyte characters in low, high byte ordering.

Syntax:

CHKSUMLOHI(string\$,bits_per_char)

Parameters:

string\$: the string whose checksum should be calculated

bits_per_char: the number of bits per character (8, 16, 24, or 32)

CHKSUMLOHI adds the character codes of all the characters in string\$. The characters can be 8, 16, 24, or 32 bits wide. The least significant byte of each character must appear first in the string.

Example:									
String:	\$10	\$08	\$35	\$F4	\$3E	\$08	\$0D	\$10	
	}		}		}		}		
Checksum:	\$0810	\$F435	\$083E	\$100D	=	\$11490			

If there are not enough bytes in the string, zeros are added internally to complete the last character.



If bits_per_char is not 8, 16, 24, or 32, CHKSUMLOHI generates an error.

■ The **CHKSUMHILO** Function

Calculates the simple checksum for a string that contains multibyte characters in high, low byte ordering.

Syntax:

CHKSUMHILO(string\$,bits_per_char)

Parameters:

string\$: the string whose checksum should be calculated

`bits_per_char`: the number of bits per character (8, 16, 24, or 32)

`CHKSUMHILO` adds the character codes of all the characters in `string$`. The characters can be 8, 16, 24, or 32 bits wide. The most significant byte of each character must appear first in the string.

Example:								
String:	\$10	\$08	\$35	\$F4	\$3E	\$08	\$0D	\$10
	}		}		}		}	
Checksum:	\$1008	\$35F4	\$3E08	\$0D10	= \$9114			

If there are not enough bytes in the string, zeros are added internally to complete the last character.



If `bits_per_char` is not 8, 16, 24, or 32, `CHKSUMHILO` generates an error.

■ The **LRC\$** Function

Creates a string containing the LRC character for a string.

Syntax:

LRC\$(`string$`)

Parameters:

`string$`: the string whose LRC should be calculated

`LRC$` XORs the character codes of all the characters in `string$`, and creates a string containing the character with the resulting character code.

To add an LRC to a string, use the following code:

```
command$=command$+LRC$(command$)
```

■ The **LRCLOHI** Function

Calculates the LRC for a string that contains multibyte characters in low, high byte ordering.

Syntax:

LRCLOHI(`string$`,`bits_per_char`)

Parameters:

`string$`: the string whose LRC should be calculated

`bits_per_char`: the number of bits per character (8, 16, 24, or 32)

`LRCLOHI` XORs the character codes of all the characters in `string$`. The characters can be 8, 16, 24, or 32 bits wide. The least significant byte of each character must appear first in the string.

Example:									
String:	\$10	\$08	\$35	\$F4	\$3E	\$08	\$0D	\$10	
LRC:	\$0810		\$F435		\$083E		\$100D		= \$E416

If there are not enough bytes in the string, zeros are added internally to complete the last character.



If `bits_per_char` is not 8, 16, 24, or 32, `LRCLOHI` generates an error.

■ The `LRCHILO` Function

Calculates the LRC for a string that contains multibyte characters in high, low byte ordering.

Syntax:

`LRCHILO(string$,bits_per_char)`

Parameters:

`string$`: the string whose LRC should be calculated
`bits_per_char`: the number of bits per character (8, 16, 24, or 32)

`LRCHILO` XORs the character codes of all the characters in `string$`. The characters can be 8, 16, 24, or 32 bits wide. The most significant byte of each character must appear first in the string.

Example:									
String:	\$10	\$08	\$35	\$F4	\$3E	\$08	\$0D	\$10	
LRC:	\$1008		\$35F4		\$3E08		\$0D10		= \$16E4

If there are not enough bytes in the string, zeros are added internally to complete the last character.

If `bits_per_char` is not 8, 16, 24, or 32, `LRCHIL0` generates an error.

■ The **CRC16** Function

Calculates the CRC-16 value for a string.

Syntax:

CRC16(string\$)

or

CRC16(string\$, initial, final)

Parameters:

string\$: the string whose CRC should be calculated

initial: the initial value of the polynomial

final: the final XOR value

`CRC16` calculates a 16 bit CRC of `string$` using the following polynomial:

$$x^{16} + x^{15} + x^2 + 1$$

`initial` is the starting value (sometimes called the seed) for the CRC calculation. The result is XORed with `final`. If you do not specify `initial` and `final`, the initial value and final XOR will both be 0.

To take the 2s complement of the result, specify a final XOR value of 65535 (\$FFFF).

If `initial` or `final` is less than 0, greater than 65535, or contains fractions, `CRC16` generates an error.

■ The **CRCCITT** Function

Calculates the CRC-CCITT value for a string.

Syntax:

CRCCITT(string\$)

or

CRCCITT(string\$, initial, final)

Parameters:

string\$: the string whose CRC should be calculated

initial: the initial value of the polynomial

final: the final XOR value

`CRCCCITT` calculates a 16 bit CRC of `string$` using the following polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

`initial` is the starting value (sometimes called the seed) for the CRC calculation. The result is XORed with `final`. If you do not specify `initial` and `final`, the initial value will be \$FFFF (65535) and the final XOR will be 0.

To take the 2s complement of the result, specify a final value of 65535 (\$FFFF).

If `initial` or `final` is less than 0, greater than 65535, or contains fractions, `CRCCCITT` generates an error.

■ The `CRC32` Function

Calculates the CRC-32 value for a string.

Syntax:

`CRC32`(`string$`)

or

`CRC32`(`string$, initial, final`)

Parameters:

`string$`: the string whose CRC should be calculated

`initial`: the initial value of the polynomial

`final`: the final XOR value

`CRC32` calculates a 32 bit CRC of `string$` using the following polynomial:

$$x^{32} + x^{26} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

`initial` is the starting value (sometimes called the seed) for the CRC calculation. The result is XORed with `final`. If you do not specify `initial` and `final`, the initial value and final XOR will both be \$FFFFFFFF (4294967295).

To take the 2s complement of the result, specify a final XOR value of 4294967295 (\$FFFFFFFF).

If `initial` or `final` is less than 0, greater than 4294967295, or contains fractions, `CRC32` generates an error.

■ The `CHKSUM$` Function

Creates a string containing the modulus 256 checksum character for a string.

Syntax:

`CHKSUM$`(`string$`)

Parameters:

`string$`: the string whose checksum should be calculated

`CHKSUM$` adds the character codes of all the characters in `string$`, and creates a string containing the character with the character code of that sum modulus 256.

To add a modulus 256 checksum to a string, use the following code:

```
command$=command$+CHKSUM$ ( command$ )
```

■ The `PRNCHKSUM$` Function

Creates a string containing the printable checksum character for a string.

Syntax:

`PRNCHKSUM$(string$)`

Parameters:

`string$`: the string whose checksum should be calculated

`PRNCHKSUM$` calculates the printable checksum for `string$`. The checksum is calculated as follows:

- Subtract 32 from each character code,
- add the results together,
- take the sum modulo 95,
- add 32 to the result.

This checksum will always be a printable character.

To add a printable checksum to a command string, use the following code:

```
command$=command$+PRNCHKSUM$ ( command$ )
```

Time and Date Functions

■ The **TIME\$** Function

Creates a string containing an SCL time value in the time and date formats specified in the *Regional Settings* control panel.

Syntax:

TIME\$(time_value)

Parameters:

time_value: the SCL time value

TIME\$ always uses 24h time, and 4 digit dates, regardless of the settings in the *Regional Settings* control panel. It will also write out the milliseconds as a 3 digit fraction of the seconds.

If time_value is not a valid time value, **TIME\$** generates an error.

■ The **MKTIME** Function

Creates an SCL time value from month, day, year, hours, minutes, and seconds.

Syntax:

MKTIME(month, day, year, hours, minutes, seconds)

Parameters:

month: the month (1=January)
day: the day of the month
year: the full year with century (e.g. 1997)
hours: the 24-hour hour (0-23)
minutes: the minutes after the hour (0-59)
seconds: the seconds after the minute (0-59)

If the parameters given do not specify an existing date and time (if you specify February 31, for example), or if the date lies before January 1, 1901, **MKTIME** generates an error.

■ The **GMT** Function

Converts a time value from the local time zone specified in the *Date-Time* control panel to Greenwich mean time.

Syntax:

GMT(local_time)

Parameters:

local_time: the local SCL time value

If local_time is not a valid time value, **GMT** generates an error.

■ The **LCTIME** Function

Converts a time value from Greenwich mean time to the local time zone specified in the *Date-Time* control panel.

Syntax:

LCTIME(GMT_time)

Parameters:

GMT_time: the Greenwich mean time SCL time value

If GMT_time is not a valid time value, LCTIME generates an error.

■ The **MON** Function

Determines the month from an SCL time value.

Syntax:

MON(time_value)

Parameters:

time_value: the SCL time value

MON returns 1 for January, 2 for February, etc.

If time_value is not a valid time value, MON generates an error.

■ The **MON\$** Function

Determines the full month name in the language specified in the *Regional Settings* control panel from an SCL time value.

Syntax:

MON\$(time_value)

Parameters:

time_value: the SCL time value

If time_value is not a valid time value, MON\$ generates an error.

■ The **MONAB\$** Function

Determines the abbreviated month name in the language specified in the *Regional Settings* control panel from an SCL time value.

Syntax:

MONAB\$(time_value)

Parameters:

time_value: the SCL time value

MONAB\$ returns "Jan" for January, "Feb" for February, etc.

If `time_value` is not a valid time value, `MONAB$` generates an error.

■ The **DAY** Function

Determines the day of the month from an SCL time value.

Syntax:

DAY(`time_value`)

Parameters:

`time_value`: the SCL time value

If `time_value` is not a valid time value, `DAY` generates an error.

■ The **YR** Function

Determines the full year with century from an SCL time value.

Syntax:

YR(`time_value`)

Parameters:

`time_value`: the SCL time value

If `time_value` is not a valid time value, `YR` generates an error.

■ The **HRS** Function

Determines the 24-hour hour from an SCL time value.

Syntax:

HRS(`time_value`)

Parameters:

`time_value`: the SCL time value

`HRS` returns the hour in 24 hour time, i.e. it returns 4 for 4am, and 16 for 4pm.

If `time_value` is not a valid time value, `HRS` generates an error.

■ The **MINS** Function

Determines the minutes after the hour from an SCL time value.

Syntax:

MINS(`time_value`)

Parameters:

`time_value`: the SCL time value

If `time_value` is not a valid time value, `MINS` generates an error.

■ The **SECS** Function

Determines the seconds after the minute from an SCL time value. It will also show the milliseconds as a 3 digit fraction of the seconds.

Syntax:

SECS(time_value)

Parameters:time\$

time_value: the SCL time value

If time_value is not a valid time value, SECS generates an error.

■ The **WKDAY** Function

Determines the day of the week from an SCL time value.

Syntax:

WKDAY(time_value)

Parameters:

time_value: the SCL time value

MON returns 1 for Monday, 2 for Tuesday, etc.

If time_value is not a valid time value, WKDAY generates an error.

■ The **WKDAY\$** Function

Determines the full week day name in the language specified in the *Regional Settings* control panel from an SCL time value.

Syntax:

WKDAY\$(time_value)

Parameters:

time_value: the SCL time value

If time_value is not a valid time value, WKDAY\$ generates an error.

■ The **WKDAYAB\$** Function

Determines the abbreviated week day name in the language specified in the *Regional Settings* control panel from an SCL time value.

Syntax:

WKDAYAB\$(time_value)

Parameters:

time_value: the SCL time value

WKDAYAB\$ returns "Mon" for Monday, "Tue" for Tuesday, etc.

If `time_value` is not a valid time value, `WKDAYAB$` generates an error.

■ The `INTVMINS$` Function

Creates a string containing a time interval, expressed as minutes and seconds, using the time separator specified in the *Regional Settings* control panel.

Syntax:

`INTVMINS$(seconds)`

Parameters:

`seconds`: the time interval, in seconds

`INTVMINS$` rounds `seconds` to the nearest second. If you want to display a counter, you should round `seconds` using the `RNDDWN` or `RNDUP` function before calling `INTVMINS$`. Use `RNDDWN` if you are counting up, and `RNDUP` if you are counting down. This will ensure that the counter changes over from one second value to the next at the correct time.

`INTVMINS$` writes the interval as minutes and seconds. If you want the interval expressed as hours, minutes, and seconds, use the `INTVHRS$` function.

If `seconds` is negative, `INTVMINS$` generates an error.

■ The `INTVHRS$` Function

Creates a string containing a time interval, expressed as hours, minutes, and seconds, using the time separator specified in the *Regional Settings* control panel.

Syntax:

`INTVHRS$(seconds)`

Parameters:

`seconds`: the time interval, in seconds

`INTVHRS$` rounds `seconds` to the nearest second. If you want to display a counter, you should round `seconds` using the `RNDDWN` or `RNDUP` function before calling `INTVHRS$`. Use `RNDDWN` if you are counting up, and `RNDUP` if you are counting down. This will ensure that the counter changes over from one second value to the next at the correct time.

`INTVHRS$` writes the interval as hours, minutes, and seconds. If you want the interval expressed as minutes and seconds only, use the `INTVMINS$` function.

If `seconds` is negative, `INTVHRS$` generates an error.

Dialog Button Callback Functions

■ The `LITEM$` Function

Gets the text of a dialog list item.

Syntax:**LITEM\$**(item_variable,index)Arguments:

item_variable: the variable associated with the list

index: the 1-based index of the list item

Use **LITEM\$** inside a dialog button callback to get the text shown in a list item. The first item has index 1.

Use **MAXLITEM** to determine the maximum list item index. Use **LITEMEXISTS%** to determine if the list item has been deleted using the **DELLITEM** command.

If it is used outside a dialog button callback, or if *item_variable* is not associated with a list, **LITEM\$** generates an error.

If *index* is smaller than 1, or larger than maximum list item index, or if the list item with the specified index has been deleted, **LITEM\$** generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **LITEMEXISTS%** Function

Determines if a dialog list has an item with the specified index.

Syntax:**LITEMEXISTS%**(item_variable,index)Arguments:

item_variable: the variable associated with the list

index: the 1-based index of the list item

Use **LITEMEXISTS%** inside a dialog button callback to determine if the list has an item with the specified index. The first item has index 1.

LITEMEXISTS% is true if an item with the specified index exists. It is false if the index is greater than the maximum list item index, or if the item has been deleted using the **DELLITEM** command.

Use **COUNTLITEMS** to determine the number of items in the list for which **LITEMEXISTS%** is true. Use **LITEM\$** to get the text of a list item.

If it is used outside a dialog button callback, or if *item_variable* is not associated with a list, **LITEMEXISTS%** generates an error.

If *index* is smaller than 1, **LITEMEXISTS%** generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **MAXLITEM** Function

Determines the maximum list item index in a dialog list.

Syntax:**MAXLITEM**(item_variable)Arguments:

item_variable: the variable associated with the list

Use MAXLITEM inside a dialog button callback to determine the largest index of all items in a dialog list. Please note that the item with that index may have been deleted using the DELLITEM command.

If you did not add any items to the list, or if you cleared the list using the CLRLITEMS command, MAXLITEM is 0.

To get the actual number of items in the list, excluding deleted ones, use the COUNTLITEMS function.

If it is used outside a dialog button callback, or if item_variable is not associated with a list, MAXLITEM generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The COUNTLITEMS Function

Counts the number of items in a dialog list.

Syntax:**COUNTLITEMS**(item_variable)Arguments:

item_variable: the variable associated with the list

Use COUNTLITEMS inside a dialog button callback to determine the number of items visible in a dialog list. COUNTLITEMS only counts existing items, items that have been deleted using the DELLITEM command are not counted.

Do not confuse this function with MAXLITEM, which returns the maximum list item index. If any items have been deleted, COUNTLITEMS is smaller than MAXLITEM.

If it is used outside a dialog button callback, or if item_variable is not associated with a list, COUNTLITEMS generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The MITEM\$ Function

Gets the text of a dialog menu item.

Syntax:**MITEM\$**(item_variable,index)Arguments:

item_variable: the variable associated with the menu

index: the 1-based index of the menu item

Use `MITEM$` inside a dialog button callback to get the text shown in a menu item. The first item has index 1.

Use `MAXMITEM` to determine the maximum menu item index. Use `MITEMEXISTS%` to determine if the menu item has been deleted using the `DELMITEM` command.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, `MITEM$` generates an error.

If `index` is smaller than 1, or larger than the maximum menu item index, or if the menu item with the specified index has been deleted, `MITEM$` generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `MITEMEXISTS%` Function

Determines if a dialog menu has an item with the specified index.

Syntax:

`MITEMEXISTS%(item_variable,index)`

Arguments:

`item_variable`: the variable associated with the menu

`index`: the 1-based index of the menu item

Use `MITEMEXISTS%` inside a dialog button callback to determine if the menu has an item with the specified index. The first item has index 1.

`MITEMEXISTS%` is true if an item with the specified index exists. It is false if the index is greater than the maximum menu item index, or if the item has been deleted using the `DELMITEM` command.

Use `COUNTMITEMS` to determine the number of items in the menu for which `MITEMEXISTS%` is true. Use `MITEM$` to get the text of a menu item.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, `MITEMEXISTS%` generates an error.

If `index` is smaller than 1, `MITEMEXISTS%` generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `MAXMITEM` Function

Determines the maximum menu item index in a dialog menu.

Syntax:

`MAXMITEM(item_variable)`

Arguments:

`item_variable`: the variable associated with the menu

Use `MAXMITEM` inside a dialog button callback to determine the largest index of all items in a dialog menu. Please note that the item with that index may have been deleted using the `DELMITEM` command.

If you did not add any items to the menu, or if you cleared the menu using the `CLRMIITEMS` command, `MAXMITEM` is 0.

To get the actual number of items in the menu, excluding deleted ones, use the `COUNTMIITEMS` function.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, `MAXMITEM` generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `COUNTMIITEMS` Function

Counts the number of items in a dialog menu.

Syntax:

`COUNTMIITEMS`(`item_variable`)

Arguments:

`item_variable`: the variable associated with the menu

Use `COUNTMIITEMS` inside a dialog button callback to determine the number of items visible in a dialog menu. `COUNTMIITEMS` only counts existing items, items that have been deleted using the `DELMITEM` command are not counted.

Do not confuse this function with `MAXMITEM`, which returns the maximum menu item index. If any items have been deleted, `COUNTMIITEMS` is smaller than `MAXMITEM`.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, `COUNTMIITEMS` generates an error.

This function is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

File Functions

■ The `FLEN` Function

Determines the length of an open file.

Syntax:

`FLEN`(`file_number`)

Parameters:

`file_number`: the file number

`FLEN` always determines the length of the file in bytes, regardless of whether `file_number` is a text or a binary file. If `file_number` is a text file, the length in bytes is

not necessarily the length in characters, since CRs are stored as a CR-LF combination in Windows. There is no way to determine the length of a text file in characters without reading all the data.

If `file_number` does not represent an open file, `FLEN` generates an error.

If `file_number` does represents a network connection, `FLEN` generates an error.

This function is not available within programs for sources, checksums, and SABus response data.

■ The `FPOS` Function

Determines the current position within an open file.

Syntax:

`FPOS(file_number)`

Parameters:

`file_number`: the file number

`FPOS` always determines the current position in bytes from the beginning of the file, regardless of whether `file_number` is a text or a binary file. If `file_number` is a text file, the position in bytes is not necessarily the position in characters, since CRs are stored as a CR-LF combination in Windows. There is no way to determine the position within a text file in characters. If you require that information, you must keep track of the file position yourself.

If `file_number` does not represent an open file, `FPOS` generates an error.

If `file_number` does represents a network connection, `FPOS` generates an error.

This function is not available within programs for sources, checksums, and SABus response data.

Register Functions

■ The `REGNAME$` Function

Returns the name of a register.

Syntax:

`REGNAME$(tag$)`

Parameters:

`tag$`: the tag of the register

If `tag$` is not the tag of a register, `REGNAME$` generates an error.

■ The **ONLOGSTR\$** Function

Returns the log string for the ON state of a register.

Syntax:

ONLOGSTR\$ (tag\$)

Parameters:

tag\$: the tag of the register

If tag\$ is not the tag of a register, ONLOGSTR\$ generates an error.

■ The **OFFLOGSTR\$** Function

Returns the log string for the OFF state of a register.

Syntax:

OFFLOGSTR\$ (tag\$)

Parameters:

tag\$: the tag of the register

If tag\$ is not the tag of a register, OFFLOGSTR\$ generates an error.

■ The **REGSTAT%** Function

Returns the ON/OFF or alarm state of a register.

Syntax:

REGSTAT% (tag\$)

Parameters:

tag\$: the tag of the register

REGSTAT% is true if the register is in its ON or alarm state, and false if it is not. If the register is masked or in its error state, REGSTAT% is always false.

Since the state of a bistate register is identical to its value, REGSTAT% with a bistate register as a parameter is equivalent to BSTVAL%.

If tag\$ is not the tag of a register, REGSTAT% generates an error.

■ The **REGMASK%** Function

Returns the mask state of a register.

Syntax:

REGMASK% (tag\$)

Parameters:

tag\$: the tag of the register

REGMASK% is true if the register is masked, and false if it is unmasked. REGMASK% is true regardless of whether the register was masked manually, automatically, or internally.

If tag\$ is not the tag of a register, REGMASK% generates an error.

■ The REGHIDDEN% Function

Returns the hidden state of a register.

Syntax:

REGHIDDEN%(tag\$)

Parameters:

tag\$: the tag of the register

REGHIDDEN% is true if the register is hidden, false if it is not.

If tag\$ is not the tag of a register, REGHIDDEN% generates an error.

■ The REGERR% Function

Returns the error state of a register.

Syntax:

REGERR%(tag\$)

Parameters:

tag\$: the tag of the register

REGERR% is true if the register is in its error state, and false if it is not. If the register is masked, REGERR% is always false.

If tag\$ is not the tag of a register, REGERR% generates an error.

■ The BSTVAL% Function

Returns the value of a bistate register.

Syntax:

BSTVAL%(tag\$)

Parameters:

tag\$: the tag of the register

BSTVAL% is true if the register is in its ON state, and false if it is not. If the register is masked or in its error state, BSTVAL% is always false.

Since the value of a bistate register is identical to its state, BSTVAL% is equivalent to REGSTAT% for bistate registers.

If tag\$ is not the tag of a bistate register, BSTVAL% generates an error.

■ The **BSTDLY** Function

Returns the response delay, in s, of a bistate register.

Syntax:

BSTDLY (tag\$)

Parameters:

tag\$: the tag of the register

If tag\$ is not the tag of a bistate register, **BSTDLY** generates an error.

■ The **DIGVAL** Function

Returns the value of a digital register.

Syntax:

DIGVAL (tag\$)

Parameters:

tag\$: the tag of the register

If the register is masked or in its error state, **DIGVAL** is always 0.

Use **DIGVAL\$** to get the name of the register's value.

If tag\$ is not the tag of a digital register, **DIGVAL** generates an error.

■ The **DIGVAL\$** Function

Returns the value name of the value of a digital register.

Syntax:

DIGVAL\$ (tag\$)

Parameters:

tag\$: the tag of the register

If the register is masked or in its error state, or if the current value of the register doesn't have a name, **DIGVAL\$** is an empty string.

Use **DIGVAL** to get the value of the register as a number.

If tag\$ is not the tag of a digital register, **DIGVAL\$** generates an error.

■ The **ANAVAL** Function

Returns the value of an analog register.

Syntax:

ANAVAL (tag\$)

or

ANAVAL (tag\$, index)

Parameters:

`tag$`: the tag of the register
`index`: the 1-based index of the value

If the register is masked or in its error state, `ANAVAL` is always 0.

`index` is the index of the value. A register with a size of one value has only a value with index 1. A register with a size of 4 values has values with indices 1, 2, 3, and 4.

If the register has a size of one value, you do not have to specify an index, as it is always 1. If the register has a size of more than one value, and you do not specify an index, then the value with the highest index will be returned.

Use `ANAGL` to access the values greater / less status. If you want to get the highest or lowest of all the register's value, use `ANAHIGH` or `ANALOW` to determine the index of the appropriate value, and then use `ANAVAL` to retrieve the value with that index.

If `tag$` is not the tag of an analog register, or if `index` is smaller than 1, larger than the size of the register, or if it contains fractions, `ANAVAL` generates an error.

■ The `ANAVAL` Function

Returns the greater / less status of an analog register's value.

Syntax:

`ANAGL`(`tag$`)

or

`ANAGL`(`tag$`, `index`)

Parameters:

`tag$`: the tag of the register
`index`: the 1-based index of the value

`ANAGL` is 0 for normal values (equal), -1 if the value's state is "less than", and 1 if it is "greater than". If the register is masked or in its error state, `ANAGL` is always 0.

`index` is the index of the value. A register with a size of one value has only a value with index 1. A register with a size of 4 values has values with indices 1, 2, 3, and 4.

If the register has a size of one value, you do not have to specify an index, as it is always 1. If the register has a size of more than one value, and you do not specify an index, then the state of the value with the highest index will be returned.

If you want to get the state of the highest or lowest of all the register's value, use `ANAHIGH` or `ANALOW` to determine the index of the appropriate value, and then use `ANAGL` to retrieve the state of the value with that index.

If `tag$` is not the tag of an analog register, or if `index` is smaller than 1, larger than the size of the register, or if it contains fractions, `ANAGL` generates an error.

■ The `ANAHIGH` Function

Returns the index of the highest of all the values of an analog register.

Syntax:**ANAHIGH**(tag\$)Parameters:

tag\$: the tag of the register

ANAHIGH does not return the actual highest value, merely its index. Use ANAHIGH to determine the index of the highest value, then retrieve the actual value using ANAVAL and ANAGL.

If the register is masked or in its error state, ANAHIGH is 0. Please remember that 0 is not a valid index for a value.

If tag\$ is not the tag of an analog register, ANAHIGH generates an error.

■ The ANALOW Function

Returns the index of the lowest of all the values of an analog register.

Syntax:**ANALOW**(tag\$)Parameters:

tag\$: the tag of the register

ANALOW does not return the actual lowest value, merely its index. Use ANALOW to determine the index of the lowest value, then retrieve the actual value using ANAVAL and ANAGL.

If the register is masked or in its error state, ANALOW is 0. Please remember that 0 is not a valid index for a value.

If tag\$ is not the tag of an analog register, ANALOW generates an error.

■ The ANAMIN Function

Returns the low limit of an analog register.

Syntax:**ANAMIN**(tag\$)Parameters:

tag\$: the tag of the register

If the register does not have a limit, ANAMIN returns a very large negative number.

If tag\$ is not the tag of an analog register, ANAMIN generates an error.

■ The ANAMAX Function

Returns the high limit of an analog register.

Syntax:**ANAMAX** (tag\$)Parameters:

tag\$: the tag of the register

If the register does not have a limit, **ANAMAX** returns a very large number.

If tag\$ is not the tag of an analog register, **ANAMAX** generates an error.

■ The STRVAL\$ Function

Returns the value of a string register.

Syntax:**STRVAL\$** (tag\$)Parameters:

tag\$: the tag of the register

If the register is masked or in its error state, **STRVAL\$** is always an empty (0 length) string.

If tag\$ is not the tag of a string register, **STRVAL\$** generates an error.

Serial Communication Functions

■ The DRVNDATA\$ Function

Returns the content of a response data element of the last command sent.

Syntax:**DRVNDATA\$** (tag\$)Parameters:

tag\$: the tag of the response data element

DRVNDATA\$ return the content of the response data element with the specified tag of the response to the last command sent using the **SENDCMD** command. **DRVNDATA\$** is an empty if no valid response was received. Use the **DRVSUCCESS%** reserved variable to determine if a valid response was received.

You can use the **DRVNDATA\$** reserved variable to access the data of the first response data element.

You cannot use **DRVNDATA\$** if you sent data to the serial port using a command other than **SENDCMD**.

If tag\$ is not the tag of a response data response element of the last command sent, **DRVNDATA\$** generates an error.

This function is not available within programs for sources, checksums, and SABus response data.

■ The **DRVNERROR** Function

Returns a numerical error code that the device returned to the last command sent.

Syntax:

DRVNERROR (tag\$)

Parameters:

tag\$: the tag of the error code

DRVNERROR returns the error code with the specified tag returned by the device to the last command sent using the **SEND CMD** command. **DRVNERROR** is 0 if a valid response was received or the command timed out. Use the **DRVERROR%** reserved variable to determine if an error response was received.

Use the **DRVNERROR\$** function to retrieve string error codes. You can use the **DRVERROR** reserved variable to access the main error code if it is a string.

You cannot use **DRVNERROR** if you sent data to the serial port using a command other than **SEND CMD**.

If tag\$ is not the tag of a numerical error code of the device of the last command sent, **DRVNERROR** generates an error.

This function is not available within programs for sources, checksums, and SABus response data.

■ The **DRVNERROR\$** Function

Returns a string error code that the device returned to the last command sent.

Syntax:

DRVNERROR\$ (tag\$)

Parameters:

tag\$: the tag of the error code

DRVNERROR\$ returns the error code with the specified tag returned by the device to the last command sent using the **SEND CMD** command. **DRVNERROR\$** is an empty string if a valid response was received or the command timed out. Use the **DRVERROR%** reserved variable to determine if an error response was received.

Use the **DRVNERROR** function to retrieve numerical error codes. You can use the **DRVERROR\$** reserved variable to access the main error code if it is numerical.

You cannot use **DRVNERROR\$** if you sent data to the serial port using a command other than **SEND CMD**.

If tag\$ is not the tag of a string error code of the device of the last command sent, **DRVNERROR\$** generates an error.

This function is not available within programs for sources, checksums, and SABus response data.

■ The **DRVENABLED%** Function

Determines if a device is enabled.

Syntax:

DRVENABLED%(port_tag\$, device_tag\$)

Parameters:

port_tag\$: the tag of the serial port the device is attached to

device_tag\$: the tag of the device

DRVENABLED% is true if the device is enabled, false if it is disabled.

In device driver programs, port_tag\$ and device_tag\$ must be empty strings (" "). Only the device the program belongs to can be accessed by device driver programs.

If port_tag\$ is not the tag of a serial port, or device_tag\$ is not the tag of a device on that port, **DRVENABLED%** generates an error.

■ The **CMDENABLED%** Function

Determines if a device command is enabled.

Syntax:

CMDENABLED%(port_tag\$, device_tag\$, command_tag\$, *command parameters*)

Parameters:

port_tag\$: the tag of the serial port the command's device is attached to

device_tag\$: the tag of the command's device

command_tag\$: the tag of the command

The tag of the command is followed by a list of parameters, separated by commas. You must specify an expression for each of the command's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

CMDENABLED% is true if the command is enabled, false if it is disabled.

In device driver programs, port_tag\$ and device_tag\$ must be empty strings (" "). Only commands of the device the program belongs to can be accessed by device driver programs.

If port_tag\$ is not the tag of a serial port, device_tag\$ is not the tag of a device on that port, or command_tag\$ is not the tag of a command in the device's driver,

CMDENABLED% generates an error.

■ The **DRVREADY%** Function

Determines if a device is ready to receive or send data.

Syntax:

DRVREADY%(port_tag\$, device_tag\$)

Parameters:

`port_tag$`: the tag of the serial port the device is attached to
`device_tag$`: the tag of the device driver the device uses

`DRVREADY%` is true if the device is enabled and communicating, false if it is disabled, or if it timed out.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only the device the program belongs to can be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, or `device_tag$` is not the tag of a device driver on that port, `DRVREADY%` generates an error.

■ The `SUSPENDED%` Function

Determines if a serial port is suspended.

Syntax:

`SUSPENDED%`(`tag$`)

Parameters:

`tag$`: the tag of the serial port

`SUSPENDED%` is true if the port is suspended, false if it is polling.

In device driver programs, `tag$` must be an empty string (" "). Only the serial port of the device the program belongs to can be accessed by device driver programs.

If `tag$` is not the tag of a serial port, `SUSPENDED%` generates an error.

Serial Device Object Functions

■ The `DRVOBJVAL` Function

Returns the value of a digital or analog serial data object.

Syntax:

`DRVOBJVAL`(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

or

`DRVOBJVAL`(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*,`index`)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object
`index`: the 1-based index of the value

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog

parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

If the data object is masked or in its error state, `DRVOBJVAL` is always 0.

`index` is the index of the value, and is only used for analog data objects. A data object with a size of one value has only a value with index 1. A data object with a size of 4 values has values with indices 1, 2, 3, and 4.

If an analog data object has a size of one value, you do not have to specify an index, as it is always 1. If the data object has a size of more than one value, and you do not specify an index, then the value with the highest index will be returned.

Use `DRVOBJGL` to access the greater / less status of an analog data object's value. If you want to get the highest or lowest of all the values of an analog data object, use `DRVOBJHIGH` or `DRVOBJLOW` to determine the index of the appropriate value, and then use `DRVOBJVAL` to retrieve the value with that index.

Use `DRVOBJVAL$` to get the value of a string data object, and `DRVOBJVAL%` to get the value of a bistate data object. You can also use `DRVOBJVAL$` to get the name of a digital data object's value.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (""). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a digital or analog data object in the device's driver, `DRVOBJVAL` generates an error.

If `object_tag$` is the tag of a digital data object and you specified an index, `DRVOBJVAL` generates an error.

If `index` is smaller than 1, larger than the size of the data object, or if it contains fractions, `DRVOBJVAL` generates an error.

■ The `DRVOBJVAL$` Function

Returns the value of a string serial data object or the name of a digital serial data object's value.

Syntax:

`DRVOBJVAL$`(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for

bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

If the data object is masked or in its error state, `DRVOBJVAL$` is always an empty string. `DRVOBJVAL$` is also an empty string if `tag$` is the tag of a digital data object and that object's value has no name.

Use `DRVOBJVAL` to get the value of an analog data object, and `DRVOBJVAL%` to get the value of a bistate data object. You can also use `DRVOBJVAL` to get the value of a digital data object as a number.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a string or digital data object in the device's driver, `DRVOBJVAL%` generates an error.

■ The `DRVOBJVAL%` Function

Returns the value of a bistate serial data object.

Syntax:

`DRVOBJVAL%`(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`DRVOBJVAL%` is true if the data object's value is ON, and false if it is OFF. If the data object is masked or in its error state, `DRVOBJVAL%` is always false.

Use `DRVOBJVAL` to get the value of a digital or analog data object, and `DRVOBJVAL$` to get the value of a digital data object. You can also use `DRVOBJVAL$` to get the name of a digital data object's value.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a bistate data object in the device's driver, `DRVOBJVAL%` generates an error.

■ The **DRVOBJGL** Function

Returns the greater / less status of an analog serial data object's value.

Syntax:

DRVOBJGL(port_tag\$, device_tag\$, object_tag\$, *object parameters*)

or

DRVOBJGL(port_tag\$, device_tag\$, object_tag\$, *object parameters*, index)

Parameters:

port_tag\$: the tag of the serial port the data object's device is attached to

device_tag\$: the tag of the data object's device

object_tag\$: the tag of the data object

index: the 1-based index of the value

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

DRVOBJGL is 0 for normal values (equal), -1 if the value's state is less than, and 1 if it is greater than. If the data object is masked or in its error state, DRVOBJGL is always 0.

index is the index of the value. A data object with a size of one value has only a value with index 1. A data object with a size of 4 values has values with indices 1, 2, 3, and 4.

If the data object has a size of one value, you do not have to specify an index, as it is always 1. If the data object has a size of more than one value, and you do not specify an index, then the state of the value with the highest index will be returned.

If you want to get the state of the highest or lowest of all the data object's value, use DRVOBJHIGH or DRVOBJLOW to determine the index of the appropriate value, and then use DRVOBJGL to retrieve the state of the value with that index.

In device driver programs, port_tag\$ and device_tag\$ must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If port_tag\$ is not the tag of a serial port, device_tag\$ is not the tag of a device on that port, or object_tag\$ is not the tag of an analog data object in the device's driver, DRVOBJGL generates an error.

If index is smaller than 1, larger than the size of the data object, or if it contains fractions, DRVOBJGL generates an error.

■ The **DRVBJHIGH** Function

Returns the index of the highest of all the values of an analog serial data object.

Syntax:

DRVBJHIGH(port_tag\$, device_tag\$, object_tag\$, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`DRVOBJHIGH` does not return the actual highest value, merely its index. Use `DRVOBJHIGH` to determine the index of the highest value, then retrieve the actual value using `DRVOBJVAL` and `DRVOBJGL`.

If the data object is masked or in its error state, `DRVOBJHIGH` is 0. Please remember that 0 is not a valid index for a value.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of an analog data object in the device's driver, `DRVOBJHIGH` generates an error.

■ The `DRVOBJLOW` Function

Returns the index of the lowest of all the values of an analog serial data object.

Syntax:

`DRVOBJLOW`(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`DRVOBJLOW` does not return the actual lowest value, merely its index. Use `DRVOBJLOW` to determine the index of the lowest value, then retrieve the actual value using `DRVOBJVAL` and `DRVOBJGL`.

If the data object is masked or in its error state, `DRVOBJLOW` is 0. Please remember that 0 is not a valid index for a value.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of an analog data object in the device's driver, `DRVOBJLOW` generates an error.

■ The `DRVOBJMASK%` Function

Returns the mask state of a serial data object.

Syntax:

DRVOBJMASK%(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`DRVOBJMASK%` is true if the data object is masked, and false if it is unmasked. `DRVOBJMASK%` is true regardless of whether the data object was masked automatically or using the `MASKDRVOBJ` command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a data object in the device's driver, `DRVOBJMASK%` generates an error.

■ The `DRVOBJERR%` Function

Returns the error state of a serial data object.

Syntax:

DRVOBJERR%(`port_tag$`,`device_tag$`,`object_tag$`, *object parameters*)

Parameters:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`DRVOBJERR%` is true if the data object is in its error state, and false if it is not. If the data object is masked, `DRVOBJERR%` is always false.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a data object in the device's driver, `DRVOBJERR%` generates an error.

Miscellaneous Functions

■ The `USRPRV%` Function

Determines if the current user is allowed to perform a certain action.

Syntax:

`USRPRV%`(`privilege_ID`)

Parameters:

`privilege_ID`: the ID of the privilege

`USRPRV%` is true if the current user is allowed to perform the specified action, false if his user level is too low.

`privilege_ID` can be one of the following values:

ID	privilege
1	acknowledge alarms
2	execute controls
3	perform port diagnostics
4	enable and disable devices
5	mask and unmask registers
6	calibrate analog registers
7	set analog register limits
8	set bistate register response times
9	change names of user-definable registers

If `privilege_ID` is not one of the values listed above, `USRPRV%` generates an error.

■ The **PVAR** Function

Gets the value of a numerical variable of the parent program.

Syntax:

PVAR(var_name\$)

Parameters:

var_name\$: the name of the variable

The parent program is the program that called this program using the **CALL**, **DRVCALL**, or **CALLRMT** command.

PVAR cannot be used to access array elements.

If var_name\$ is not a valid name for a numerical variable, or if it is the name of a reserved SCL keyword, **PVAR** generates an error.

If the variable specified by var_name\$ is an array, **PVAR** generates an error.

This function is only available within child programs.

■ The **PVAR\$** Function

Gets the value of a string variable of the parent program.

Syntax:

PVAR\$(var_name\$)

Parameters:

var_name\$: the name of the variable

The parent program is the program that called this program using the **CALL**, **DRVCALL**, or **CALLRMT** command.

PVAR\$ cannot be used to access array elements.

If var_name\$ is not a valid name for a string variable, or if it is the name of a reserved SCL keyword, **PVAR\$** generates an error.

If the variable specified by var_name\$ is an array, **PVAR\$** generates an error.

This function is only available within child programs.

■ The **PVAR%** Function

Gets the value of a Boolean variable of the parent program.

Syntax:

PVAR%(var_name\$)

Parameters:

var_name\$: the name of the variable

The parent program is the program that called this program using the **CALL**, **DRVCALL**, or **CALLRMT** command.

PVAR% cannot be used to access array elements.

If var_name\$ is not a valid name for a Boolean variable, or if it is the name of a reserved SCL keyword, PVAR% generates an error.

If the variable specified by var_name\$ is an array, PVAR% generates an error.

This function is only available within child programs.

Special Purpose Functions

■ The DRVPRM Function

Returns the value of a digital or analog parameter of the device that the program belong to.

Syntax:

DRVPRM(tag\$)

Parameters:

tag\$: the tag of the parameter

DRVPRM retrieves the value of a digital or analog device driver parameter. You can use this function to get the address of the device that the program belongs to, for example.

Use DRVPRM\$ to access string parameters, and DRVPRM% to access bistate parameters. You can also use DRVPRM\$ to access the name of a digital parameter's value.

if tag\$ is not the tag of a digital or analog parameter of the device driver the program belong to, DRVPRM generates an error.

This function is only available within device driver programs.

■ The DRVPRM\$ Function

Returns the value of a string parameter or the name of the value of a digital parameter of the device that the program belong to.

Syntax:

DRVPRM\$(tag\$)

Parameters:

tag\$: the tag of the parameter

DRVPRM\$ retrieves the value of a string device driver parameter, or the name of the value of a digital device driver parameter. You can use this function to get the address of the device that the program belongs to, for example.

If tag\$ is the tag of a digital parameter, but that parameter's value has no name, DRVPRM\$ is an empty string.

Use DRVPRM to access analog parameters, and DRVPRM% to access bistate parameters. You can also use DRVPRM to access the value of a digital parameter as a number.

if `tag$` is not the tag of a string or digital parameter of the device driver the program belong to, `DRVPRM$` generates an error.

This function is only available within device driver programs.

■ The **DRVPRM** Function

Returns the value of a bistate parameter of the device that the program belong to.

Syntax:

DRVPRM%(`tag$`)

Parameters:

`tag$`: the tag of the parameter

`DRVPRM%` retrieves the value of a bistate device driver parameter. `DRVPRM%` is true if the parameter is ON, and false if it is OFF. You can use this function to get options of the device that the program belong to, for example.

Use `DRVPRM` to access digital or analog parameters, and `DRVPRM$` to access digital parameters. You can also use `DRVPRM$` to access the name of a digital parameter's value.

if `tag$` is not the tag of a bistate parameter of the device driver the program belong to, `DRVPRM%` generates an error.

This function is only available within device driver programs.

■ The **BUFFER** Function

Gets a single data buffer byte as a number.

Syntax:

BUFFER(`n`)

Parameters:

`n`: the 1-based index of the byte

`BUFFER` retrieves single bytes from the data that is to be evaluated. Use the `BUFFER$` reserved variable to retrieve the entire buffer as a string.

The first byte in the buffer has index 1. If `n` is greater than the number of bytes in the buffer, `BUFFER` returns -1.

if `n` is smaller or equal to 0, `BUFFER` generates an error.

This function is only available within programs for processor sources or checksums.

■ The **TRIGGERPRM** Function

Returns the value of a digital or analog parameter of the serial data object whose value this program is determining.

Syntax:

TRIGGERPRM(`tag$`)

Parameters:

tag\$: the tag of the parameter

TRIGGERPRM is used in processor and summary sources to retrieve the value of a digital or analog parameter of the data object in whose source this program is used.

Use TRIGGERPRM\$ to access string parameters, and TRIGGERPRM% to access bistate parameters. You can also use TRIGGERPRM\$ to access the name of a digital parameter's value.

Use the TRIGGER\$ reserved variable to access the name of the data object.

if tag\$ is not the tag of a digital or analog parameter of the data object this program is evaluating, TRIGGERPRM generates an error.

This function is only available programs for processor and summary sources.

■ The TRIGGERPRM\$ Function

Returns the value of a string parameter or the name of the value of a digital parameter of the serial data object whose value this program is determining.

Syntax:

TRIGGERPRM\$ (tag\$)

Parameters:

tag\$: the tag of the parameter

TRIGGERPRM\$ is used in processor and summary sources to retrieve the value of a string parameter or the name of the value of a digital parameter of the data object in whose source this program is used.

If tag\$ is the tag of a digital parameter, but that parameter's value has no name, TRIGGERPRM\$ is an empty string.

Use TRIGGERPRM to access analog parameters, and TRIGGERPRM% to access bistate parameters. You can also use TRIGGERPRM to access the value of a digital parameter as a number.

Use the TRIGGER\$ reserved variable to access the name of the data object.

if tag\$ is not the tag of a string or digital parameter of the data object this program is evaluating, TRIGGERPRM\$ generates an error.

This function is only available programs for processor and summary sources.

■ The TRIGGERPRM% Function

Returns the value of a bistate parameter of the serial data object whose value this program is determining.

Syntax:

TRIGGERPRM% (tag\$)

Parameters:

tag\$: the tag of the parameter

TRIGGERPRM% is used in processor and summary sources to retrieve the value of a bistate parameter of the data object in whose source this program is used. TRIGGERPRM% is true if the parameter is ON, and false if it is OFF.

Use TRIGGERPRM to access digital or analog parameters, and TROGGERPRM\$ to access digital parameters. You can also use TRIGGERPRM\$ to access the name of a digital parameter's value.

Use the TRIGGER\$ reserved variable to access the name of the device object.

if tag\$ is not the tag of a bistate parameter of the data object this program is evaluating, TRIGGERPRM generates an error.

This function is only available programs for processor and summary sources.

■ The RTSPRM\$ Function

returns an RTS parameter.

Syntax:

RTSPRM\$(n)

Parameters:

n: the index of the parameter

The first parameter in has index 1. If n is greater than the number of parameters, RTSPRM\$ returns an empty string.

if n is smaller or equal to 0, RTSPRM\$ generates an error.

This function is only available within RTS controls.

Legacy Object Functions

■ The MSGENABLED% Function

Determines if a legacy device driver message is enabled.

Syntax:

MSGENABLED%(port_tag\$,device_tag\$,message_tag\$)

Parameters:

port_tag\$: the tag of the serial port the message's device is attached to

device_tag\$: the tag of the device of the message

message_tag\$: the tag of the message

MSGENABLED% is true if the message is enabled, false if it is disabled.

If port_tag\$ is not the tag of a serial port, device_tag\$ is not the tag of a device on the port that uses a legacy device driver, or message_tag\$ is not the tag of a message in that driver, MSGENABLED% generates an error.

■ The **PARAM** Function

Returns the value of a legacy parameter as a number.

Syntax:

PARAM(tag\$)

Parameters:

tag\$: the tag of the parameter

PARAM returns a number written out in decimal form in the parameter's value.

PARAM ignores any characters in the value that appear after the number. If the value does not contain a number, or if there are characters other than spaces before the number, **PARAM** returns 0.

If the parameter has never been set, **PARAM** uses its default value.

If tag\$ is not the tag of a parameter, **PARAM** generates an error.

■ The **PARAM\$** Function

Returns the value of a legacy parameter as a string.

Syntax:

PARAM\$(tag\$)

Parameters:

tag\$: the tag of the parameter

PARAM\$ returns an exact copy of the value of the parameter. If the parameter has never been set, **PARAM\$** returns its default value.

If tag\$ is not the tag of a parameter, **PARAM\$** generates an error.

■ The **PARAM%** Function

Returns the value of a legacy parameter as a Boolean value.

Syntax:

PARAM%(tag\$)

Parameters:

tag\$: the tag of the parameter

PARAM% returns true if the parameter's value is "ON", otherwise, false. **PARAM%** is not case sensitive, it returns true for "on", "On", and "oN" as well.

If the parameter has never been set, **PARAM%** uses its default value.

If tag\$ is not the tag of a parameter, **PARAM%** generates an error.

Obsolete Functions

These functions are obsolete and should not be used:

- The `MITEQ$` Function (Use the `PRNCHKSUM$` function instead.)
- The `LEFMT$` Function (Use the `HILOFMT$` function instead.)
- The `BEFMT$` Function (Use the `LOHIFMT$` function instead.)
- The `SLEVAL` Function (Use the `SHILOVAL` function instead.)
- The `SBEVAL` Function (Use the `SLOHIVAL` function instead.)
- The `LEVAL` Function (Use the `HILOVAL` function instead.)
- The `BEVAL` Function (Use the `LOHIVAL` function instead.)

COMMAND REFERENCE

Flow Control Commands

■ The **GOTO** Command

Syntax:

GOTO(*line_number*)

Arguments:

line_number: the line number

If *line_number* is not an existing line number, **GOTO** generates an error.

■ The **GOSUB** Command

Jumps to a subroutine.

Syntax:

GOSUB(*line_number*)

Arguments:

line_number: the line number at which the subroutine starts

If *line_number* is not an existing line number, **GOSUB** generates an error.

■ The **ON...GOTO** Command

Jumps to one of a list of line numbers, depending on the value of an index.

Syntax:

ON *index* **GOTO** *line_number_1*, *line_number_2*, *line_number_3*, *etc.*

Arguments:

index: the index of the line number to jump to

line_number_1, *etc.*: the line numbers

You can specify any number of line numbers. If *index* is 1, **ON...GOTO** jumps to *line_number_1*, if *index* is 2, it jumps to *line_number_2*, if *index* is 3, it jumps to *line_number_3*, and so on. If *index* is smaller than one, greater than the number of line numbers specified, or not an integer, **ON...GOTO** does not jump, and execution continues at the next statement.

If the number at position *index* is not an existing line number, **ON...GOTO** generates an error.

■ The **ON...GOSUB** Command

Jumps to one of a list of subroutines, depending on the value of an index.

Syntax:

ON *index* **GOSUB** *line_number_1*, *line_number_2*, *line_number_3*, *etc.*

Arguments:

index: the index of the line number to jump to

line_number_1, *etc.*: the line numbers at which the subroutines start

You can specify any number of line numbers. If *index* is 1, **ON...GOSUB** jumps to *line_number_1*, if *index* is 2, it jumps to *line_number_2*, if *index* is 3, it jumps to *line_number_3*, and so on. If *index* is smaller than one, greater than the number of line numbers specified, or not an integer, **ON...GOSUB** does not jump, and execution continues at the next statement.

If the number at position *index* is not an existing line number, **ON .GOSUB** generates an error.

■ The **RETURN** Command

Returns from a subroutine.

Syntax:

RETURN

If execution is not within a subroutine, **RETURN** generates an error.

■ The **IF** Command

Executes a single command or a block of code only if a Boolean expression is true.

Syntax:

IF *condition%* **THEN** *command*

or

IF *condition%* **THEN**

Arguments:

condition%: the Boolean expression

command: the command to execute if *condition%* is true

The first syntax variant is used for single command conditions. The command will only be executed if *condition%* evaluates to true.

The second syntax variant starts an **IF-THEN-ENDIF** block. An **IF-THEN-ENDIF** block allows you to specify a block of commands and assignments that will only be executed if the Boolean expression is true. The block is terminated with the **ENDIF** keyword:

```
IF condition% THEN
    command 1
    command 2
```

```

    command 3
    etc.
ENDIF

```

The commands will only be executed if `condition%` evaluates to true.

Note how `command 1` is not on the same line as the `THEN` keyword. If you place the command on the same line, you must separate it from the `THEN` keyword using a colon, or the parser will interpret it as a single command condition.

See Conditional Statements on page 19 for details.

■ The **ELSEIF** Command

Executes a block of code only if a Boolean expression is true, and a previous conditional statement was false.

Syntax:

```
ELSEIF condition% THEN
```

Arguments:

`condition%`: the Boolean expression

ELSEIF starts an additional block of commands and assignments with an additional condition within an **IF-THEN-ENDIF** block:

```

IF condition_1% THEN
    ...command 1
    ...command 2
    ...command 3
    etc.
ELSEIF condition_2% THEN
    ...command 4
    ...command 5
    ...command 6
    etc.
ENDIF

```

If `condition_1%` is true, commands 1, 2, 3, etc. will be executed.

If `condition_1%` is false, but `condition_2%` is true, commands 4, 5, 6, etc. will be executed.

You can have any number of **ELSEIF** blocks within an **IF-THEN-ENDIF** block. You can also combine **ELSEIF** and **ELSE** blocks.

See Conditional Statements on page 19 for details.

■ The **ELSE** Command

Executes a block of code only if a previous conditional statement was false.

Syntax:**ELSE**

ELSE specifies a second block of commands and assignments in an IF-THEN-ENDIF block that will be executed if the IF condition is false:

```

IF condition% THEN
...command 1
...command 2
...command 3
  etc.
ELSE
...command 4
...command 5
...command 6
  etc.
ENDIF

```

The commands 1, 2, 3, etc. will be executed if condition% evaluates to true. If condition% is false, commands 4, 5, 6, etc. will be executed instead.

See Conditional Statements on page 19 for details.

■ The **ENDIF** Command

Ends an IF-THEN-ENDIF block.

Syntax:**ENDIF**

See Conditional Statements on page 19 for details.

■ The **WHILE** Command

Starts a WHILE-DO loop.

Syntax:

```
WHILE continue_contition% DO command
```

or

```
WHILE continue_contition% DO
```

Arguments:

continue_condition%: the Boolean expression that controls the execution of the loop

The first syntax variant is used for a single command loop. The command will be executed as long as continue_contition% remains true.

The second syntax variant is used to start a WHILE-DO-ENDDO block:

```

WHILE continue_contition% DO
  command 1
  command 2

```

```

    command 3
    etc.
ENDDO

```

Note how the first command in the `WHILE-DO-ENDDO` block is not on the same line as the `DO` keyword. If you place them on the same line, you must separate them using a colon, or the parser will interpret it as a single command loop.

If `continue_contition%` is false initially, the command or commands are never executed.

See Loops on page 22 for details.

■ The **ENDDO** Command

Ends a `WHILE-DO-ENDO` block.

Syntax:
ENDDO

See Loops on page 22 for details.

■ The **REPEAT** Command

Starts a `REPEAT-UNTIL` loop.

Syntax:
REPEAT

`REPEAT` starts a `REPEAT-UNTIL` loop:

```

REPEAT
    command 1
    command 2
    command 3
    etc.
UNTIL stop_condition%

```

Everytime the program reaches the `UNTIL` statement, it evaluates `stop_condition%`. If `stop_contition%` is false, it jumps back to the `REPEAT` statement. If `stop_condition%` is true, it goes on to the next command or assignment after the `UNTIL`.

See Loops on page 22 for details.

■ The **UNTIL** Command

Ends a `REPEAT-UNTIL` loop.

Syntax:
UNTIL *stop_condition%*

Arguments:

`stop_condition%`: the Boolean expression that controls the execution of the loop

See Loops on page 22 for details.

■ The FOR Command

Start a FOR-NEXT loop.

Syntax:

```
FOR counter = initial_value TO final_value
```

or

```
FOR counter = initial_value TO final_value STEP step_size
```

Arguments:

counter: the variable used to count
initial_value: the initial value of the variable
final_value: the value to which to count
step_size: the increments in which to count

A FOR-NEXT loop allows you to specify a block that will be executed a specific number of times. This is done by “counting” from one number to another. You must specify a numerical variable or array element that the program uses to count. a simple FOR-NEXT loop looks like this:

```
FOR counter = initial_value TO final_value STEP step_size  
    command 1  
    command 2  
    command 3  
    etc.  
NEXT counter
```

(You can also just write NEXT instead of NEXT counter)

The program will start “counting” by setting counter to initial_value. It will then execute the block of commands. Once it reaches the NEXT statement, it will add step_size to counter and execute the block again. If you do not specify a step size, the program will count in increments of 1 or -1, depending on whether final_value or initial_value is greater.

The block will be executed once with counter set to each value between initial_value and final_value, inclusive. You can, of course, access counter within the block like any other variable.

If the sign of step_size would cause the program to count in the wrong direction (i.e. away from final_value instead of towards it), the block of commands is executed once with counter set to initial_value.

After the loop is finished, counter will be one step-size *beyond* the value it had when the loop last executed. For the statement

```
FOR counter = 2.1 TO 8.5
```

counter will have the value 9.1 after the loop is done.

If step_size is 0, the parser generates an error.

See Loops on page 22 for details.

■ The **NEXT** Command

Ends a **FOR-NEXT** loop.

Syntax:

NEXT

or

NEXT counter

Arguments:

counter: the variable used in the **FOR** statement

If the variable is not the same as that specified in the last **FOR** statement, the parse generates an error.

See Loops on page 22 for details.

■ The **END** Command

Ends the program.

Syntax:

END

User Message Commands

■ The **PROMPT** Command

Appends text to the internally maintained output buffer.

Syntax:

PROMPT *list of expressions*

See Messages to the User on page 25 for details on the output buffer.

PROMPT is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are placed into the output buffer directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the buffer.

PROMPT places the expressions of each **PROMPT** command together on one line. The expressions of the next **PROMPT** command will be placed on a separate line, unless you do one of the following:

To place the output of the *next* **PROMPT** command immediately after the output of this one, put a semicolon (;) at the end of *this* **PROMPT** command.

To place the output of the *next* PROMPT command on the same line as the output of this one but separated by a tab, put a comma at the end of *this* PROMPT command.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The INFO Command

Displays the internally maintained output buffer in a message window, optionally appending additional expressions.

Syntax:

INFO

or

INFO list of expressions

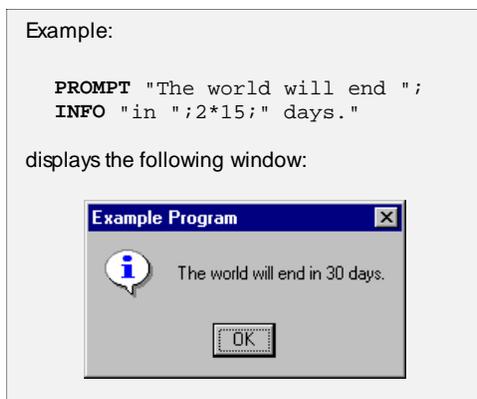
See Messages to the User on page 25 for details on the output buffer.

INFO may be followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons, which will be placed into the output buffer. String expressions are placed into the output buffer directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the buffer.

The INFO dialog has only an OK button. The output buffer is emptied by the INFO command.

The INFO dialog has a different icon than the PRINT and ERRMSG commands, and Windows plays a different sound. Please refer to When to Use Which Command in *Messages to the User* on page 26 for information on when to use which command.



This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **PRINT** Command

Displays the internally maintained output buffer in a message window, optionally appending additional expressions.

Syntax:

PRINT

or

PRINT list of expressions

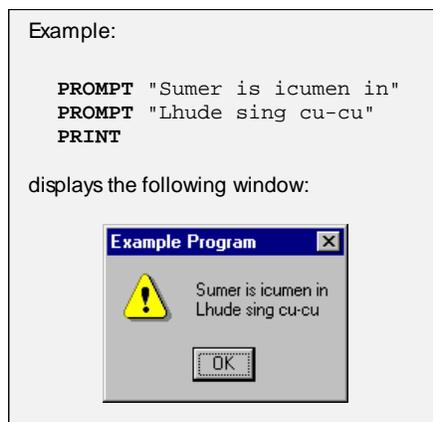
See Messages to the User on page 25 for details on the output buffer.

PRINT may be followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons, which will be placed into the output buffer. String expressions are placed into the output buffer directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the buffer.

The **PRINT** dialog has only an OK button. The output buffer is emptied by the **PRINT** command.

The **PRINT** dialog has a different icon than the **INFO** and **ERRMSG** commands, and Windows plays a different sound. Please refer to When to Use Which Command in *Messages to the User* on page 26 for information on when to use which command.



Note: If **PRINT** is terminated with a semicolon or comma, it acts like the **PROMPT** command, and will not output anything to the screen. This usage of **PRINT** is obsolete and should not be used.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **ERRMSG** Command

Displays the internally maintained output buffer in a message window, optionally appending additional expressions.

Syntax:

ERRMSG

or

ERRMSG list of expressions

See Messages to the User on page 25 for details on the output buffer.

ERRMSG may be followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons, which will be placed into the output buffer. String expressions are placed into the output buffer directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the buffer.

The **ERRMSG** dialog has only an OK button. The output buffer is emptied by the **ERRMSG** command.

The **ERRMSG** dialog has a different icon than the **INFO** and **PRINT** commands, and Windows plays a different sound. Please refer to When to Use Which Command in *Messages to the User* on page 26 for information on when to use which command.



This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **CONFIRM** Command

Displays the internally maintained output buffer in a dialog, giving the user the option to proceed or cancel.

Syntax:

CONFIRM

or

CONFIRM result_variable%

Arguments:

`result_variable%`: a variable that is to receive the user's choice

See Messages to the User on page 25 for details on the output buffer.

The `CONFIRM` dialog has an OK and a Cancel button. The output buffer is emptied by the `CONFIRM` command.

If you do not specify a result variable, `CONFIRM` will end the program if the user chooses to cancel the operation. If you do specify a result variable, it will be set to true if the user chooses to proceed, and false if he chooses to cancel.



This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `ASK` Command

Displays the internally maintained output buffer in a dialog, giving the user the option to answer "Yes" or "No."

Syntax:

ASK

or

ASK `result_variable%`

Arguments:

`result_variable%`: a variable that is to receive the user's choice

See Messages to the User on page 25 for details on the output buffer.

The `ASK` dialog has a Yes and a No button. The output buffer is emptied by the `ASK` command.

If you do not specify a result variable, `ASK` will end the program if the user answers "No." If you do specify a result variable, it will be set to true if the user answers "Yes," and false if he answers "No."



This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

Dialog Commands

■ The **DIALOG** Command

Shows the current dialog.

Syntax:
DIALOG

or

DIALOG result_variable

or

DIALOG result_variable, line_number

Arguments:

result_variable: the result variable

line_number: the line number where the button callback routine starts

DIALOG shows the dialog previously constructed using dialog commands. See Dialogs on page 28 for more details. **DIALOG** deletes the current dialog.

If you specify a result variable, the variable will contain the button number of the button the user pressed to close the dialog.

If you do not specify a result variable, the program will be aborted if the user presses the cancel button. If the user presses any other button, the program will continue.

If you specify a result variable, you can also specify a button callback line number.

Whenever the user presses a button other than the Cancel button, SCL jumps to the line number as if it encountered a **GOSUB** command. You can look at the result variable to see what button the user pressed. All the variables associated with the items are updated to reflect the user's entries.

You return from the button callback using the **RETURN** command.

If you want the dialog to be closed when you return, do not modify the result variable. If you want the dialog to stay up, set the result variable to 0. If you set the result variable to any other value but 0, the dialog will be closed, and the result variable will retain that value when the program continues after the `DIALOG` statement.

If you set the result variable to 0, all dialog items will be updated to reflect any changes you made in their variables, and the dialog will stay up.

The button callback will be called again once the user presses another button.

The button callback is not called if the user presses the Cancel button.

if `line_number` is not an existing line number, `DIALOG` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `DLGTITLE` Command

Set the title of the current dialog.

Syntax:

DLGTITLE title\$

Arguments:

title\$: the new title

Use `DLGTITLE` to show a custom title in the title bar of the dialog. Applies only to the dialog currently being constructed. If you do not call `DLGTITLE` for a particular dialog, the title bar of the dialog will contain the name of the program.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `DLGTEXT` Command

Adds a static text field to the current dialog.

Syntax:

DLGTEXT list of expressions

Use `DLGTEXT` to add static text to a dialog. Static text cannot be edited by the user.

`DLGTEXT` is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are sent directly. Numbers are sent in decimal format, using exponential notation whenever necessary. Boolean values are sent as "true" or "false."

Expressions separated by a semicolon are sent immediately next to each other, expressions separated by a comma are sent with a tab separating them.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **DLGLINE** Command

Adds a horizontal separator line to the current dialog.

Syntax:

DLGLINE

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **STREDIT** Command

Adds a text entry field to the current dialog. The field does not allow an empty entry.

Syntax:

STREDIT `prompt$,string_variable$`

or

STREDIT `prompt$,string_variable$,width`

or

STREDIT `prompt$,string_variable$,units$`

or

STREDIT `prompt$,string_variable$,units$,width`

Arguments:

`prompt$`: the prompt
`string_variable$`: the variable associated with the edit field
`units$`: a string representing the units
`width`: the width of the entry field

Use this command to add an edit field that lets the user enter a non-empty string.

`prompt$` is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character ("`_`") after the appropriate character in the prompt.

`string_variable$` has to be a string variable. The edit field is initialized with the content of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

`units$` is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`width` is an optional width of the edit field. If you specify a width, the edit field will hold approximately that many characters. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what characters they are. If you don't specify a width, the edit field will extend across the entire dialog.

`STREDIT` will disable all buttons added with the `BUTTON` command if it contains no characters. To allow the user to leave the edit field blank, use the `STREDIT0` command.

If the user is supposed to enter a password, use the `PWDEDIT` command instead. The `PWDEDIT` command does not show characters as they are typed, which is necessary to hide the password from casual onlookers.

If `width` is negative or zero, or if it contains fractions, `STREDIT` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `STREDIT0` Command

Adds a text entry field to the current dialog. The field allows an empty entry.

Syntax:

STREDIT0 `prompt$,string_variable$`

or

STREDIT0 `prompt$,string_variable$,width`

or

STREDIT0 `prompt$,string_variable$,units$`

or

STREDIT0 `prompt$,string_variable$,units$,width`

Arguments:

`prompt$:` the prompt
`string_variable$:` the variable associated with the edit field
`units$:` a string representing the units
`width:` the width of the entry field

Use this command to add an edit field that lets the user enter a string that may be empty.

`prompt$` is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character ("`_`") after the appropriate character in the prompt.

`string_variable$` has to be a string variable. The edit field is initialized with the content of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

`units$` is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`width` is an optional width of the edit field. If you specify a width, the edit field will hold approximately that many characters. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what characters they are. If you don't specify a width, the edit field will extend across the entire dialog.

`STREDIT0` allows the user to leave the edit field blank. If you want to force the user to enter something in the field, use the `STREDIT` command.

If the user is supposed to enter a password, use the `PWDEDIT0` command instead. The `PWDEDIT0` command does not show characters as they are typed, which is necessary to hide the password from casual onlookers.

If `width` is negative or zero, or if it contains fractions, `STREDIT0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The PWEDIT Command

Adds a password entry field to the current dialog. The field does not allow an empty entry.

Syntax:

PWEDIT *prompt\$,string_variable\$*

or

PWEDIT *prompt\$,string_variable\$,width*

or

PWEDIT *prompt\$,string_variable\$,units\$*

or

PWEDIT *prompt\$,string_variable\$,units\$,width*

Arguments:

prompt\$: the prompt
string_variable\$: the variable associated with the edit field
units\$: a string representing the units
width: the width of the entry field

Use this command to add an edit field that lets the user enter a non-empty password string.

prompt\$ is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for *prompt\$*.
- To have the prompt on a separate line, use a string that end with a return character (RET\$)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character ("_") after the appropriate character in the prompt.

string_variable\$ has to be a string variable. The edit field is initialized with the content of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

units\$ is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`width` is an optional width of the edit field. If you specify a width, the edit field will hold approximately that many characters. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what characters they are. If you don't specify a width, the edit field will extend across the entire dialog.

`PWDEDIT` will disable all buttons added with the `BUTTON` command if it contains no characters. To allow the user to leave the edit field blank, use the `PWDEDIT0` command.

The `PWDEDIT` command shows characters as asterisks ("`*`") as they are typed, which is necessary to hide the password from casual onlookers. For regular string entry, use the `STREDIT` command.

If `width` is negative or zero, or if it contains fractions, `PWDEDIT` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `PWDEDIT0` Command

Adds a password entry field to the current dialog. The field allows an empty entry.

Syntax:

```
PWDEDIT0 prompt$,string_variable$
```

or

```
PWDEDIT0 prompt$,string_variable$,width
```

or

```
PWDEDIT0 prompt$,string_variable$,units$
```

or

```
PWDEDIT0 prompt$,string_variable$,units$,width
```

Arguments:

```
prompt$:           the prompt
string_variable$:  the variable associated with the edit field
units$:           a string representing the units
width:            the width of the entry field
```

Use this command to add an edit field that lets the user enter a password string that may be empty.

`prompt$` is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character ("`_`") after the appropriate character in the prompt.

`string_variable$` has to be a string variable. The edit field is initialized with the content of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

`units$` is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`width` is an optional width of the edit field. If you specify a width, the edit field will hold approximately that many characters. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what characters they are. If you don't specify a width, the edit field will extend across the entire dialog.

`PWDEDIT0` allows the user to leave the edit field blank. If you want to force the user to enter a password in the field, use the `PWDEDIT` command.

The `PWDEDIT0` command shows characters as asterisks ("`**`") as they are typed, which is necessary to hide the password from casual onlookers. For regular string entry, use the `STREDIT0` command.

If `width` is negative or zero, or if it contains fractions, `PWDEDIT0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **NUMEDIT** Command

Adds a number entry field to the current dialog.

Syntax:

NUMEDIT `prompt$,numerical_variable`

or

NUMEDIT `prompt$,numerical_variable,width`

or

NUMEDIT `prompt$,numerical_variable,units$`

or

NUMEDIT `prompt$,numerical_variable,units$,width`

Arguments:

`prompt$`: the prompt
`numerical_variable`: the variable associated with the edit field
`units$`: a string representing the units
`width`: the width of the entry field

Use this command to add an edit field that lets the user enter a number.

`prompt$` is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character ("`_`") after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. The edit field is initialized to the value of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

`units$` is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`width` is an optional width of the edit field. If you specify a width, the edit field will hold approximately that many characters. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what char-

acters they are. If you don't specify a width, the edit field will extend across the entire dialog.

NUMEDIT allows the user to enter any number. If you want to restrict the entry to numbers without fractions, use the INTEDIT command.

If width is negative or zero, or if it contains fractions, NUMEDIT generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The INTEDIT Command

Adds an integer entry field to the current dialog. The entry field has a thumbwheel.

Syntax:

INTEDIT prompt\$,numerical_variable

or

INTEDIT prompt\$,numerical_variable,minimum

or

INTEDIT prompt\$,numerical_variable,minimum,maximum

or

INTEDIT prompt\$,numerical_variable,minimum,maximum,width

or

INTEDIT prompt\$,numerical_variable,units\$

or

INTEDIT prompt\$,numerical_variable,units\$,minimum

or

INTEDIT prompt\$,numerical_variable,units\$,minimum,maximum

or

INTEDIT prompt\$,numerical_variable,units\$,minimum,maximum,width

Arguments:

prompt\$: the prompt

numerical_variable: the variable associated with the edit field

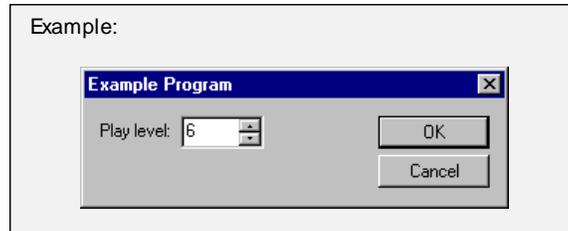
units\$: a string representing the units

minimum: the lower limit of the thumbwheel

maximum: the upper limit of the thumbwheel

width: the width of the entry field

Use this command to add an edit field that lets the user enter a number without fractions. The control has a thumbwheel to allow the user to easily increment and decrement the value. A thumbwheel consists of two small arrows (one up and one down) at the right of the edit field.



`prompt$` is a string that tells the user what he should enter in the edit field. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the edit field, use a string that does not end in a return character

To assign a keyboard shortcut to the edit field, put an underscore character (“_”) after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. The edit field is initialized to the value of the variable, and the variable is set to the user entry when the user presses a button other than the Cancel button.

`units$` is optional string that is placed to the right of the edit field. You can use it to tell the user what units the data that he enters is in.

`minimum` and `maximum` are the limits that the user can set the value to using the thumbwheel. If you do not specify one of the limits, the user can increase or decrease indefinitely. Please note that the user can *type* any number he wants. `minimum` and `maximum` only apply to the thumbwheel. `INTEDIT` does not guarantee that `numerical_variable` will not be outside the specified range, you must check that yourself in the dialog button callback routine.

`width` is an optional width of the edit field. If you specify a width, the edit field will be wide enough to hold approximately that many characters, less the width of the thumbwheel. Please note that since not all characters are the same width, how many characters will fit in an edit field depends greatly on what characters they are. If you don't specify a width, the edit field will extend across the entire dialog.

INTEDIT only allows the user to enter numbers without fractions. If you want to enable the user to enter any number, use the NUMEDIT command instead.

If the value of `numerical_variable` contains fractions, INTEDIT generates an error.

If `maximum` is less than `minimum`, INTEDIT generates an error.

If `width` is negative or zero, or if it contains fractions, INTEDIT generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The CHKBOX Command

Adds a check box to the current dialog.

Syntax:

CHKBOX `Boolean_variable`, `label$`

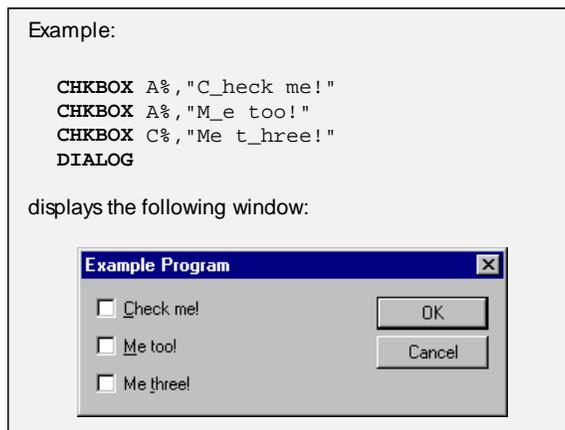
Arguments:

`Boolean_variable`: the variable associated with the check box

`label$`: the label of the check box

`label$` is the text that appears next to the check box. To assign a keyboard shortcut to the check box, put an underscore character (" _ ") after the appropriate character in the prompt.

`Boolean_variable` has to be a Boolean variable. The check box is checked initially if the value of the variable is true. The variable is set to true if the check box is checked when the user presses a button other than the Cancel button, false otherwise.



This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The LIST Command

Adds an unsorted auto-width list to the current dialog. The list requires a user selection.

Syntax:

LIST *prompt\$,numerical_variable*

or

LIST *prompt\$,numerical_variable,item1\$,item2\$,item3\$, etc.*

or

LIST *prompt\$,numerical_variable,height*

or

LIST *prompt\$,numerical_variable,height,item1\$,item2\$,item3\$, etc.*

Arguments:

prompt\$: the prompt

numerical_variable: the variable associated with the list

height: the number of items the list can display without scrolling

item1\$,item2\$,item3\$, etc.: items to be placed in the list

Use this command to add a list that requires the user to select an item.

prompt\$ is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character (“_”) after the appropriate character in the prompt.

numerical_variable has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button.

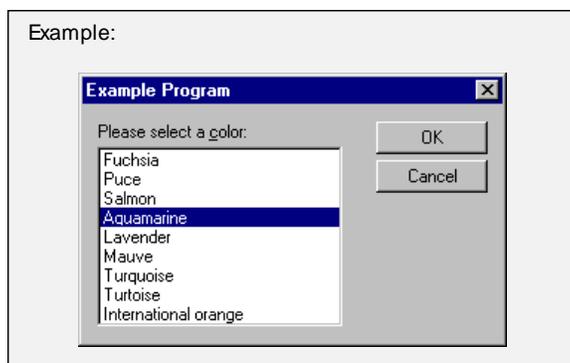
The list will be at least wide enough to hold the longest of the items you add. To specify a specific width for the list, use the **LISTW** command.

height is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the **LIST** command, or you can add items later using the **LITEM** command.

LIST will disable all buttons added with the **BUTTON** command if no item is selected. To allow the user to proceed without selecting an item, use the **LIST0** command.

If you want the items in the list to be sorted alphabetically, use the **SLIST** command.



If the value of `numerical_variable` is negative, or if it contains fractions, `LIST` generates an error.

If `height` is negative or zero, or if it contains fractions, `LIST` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `LISTW` Command

Adds an unsorted fixed-width list to the current dialog. The list requires a user selection.

Syntax:

`LISTW` `prompt$`, `numerical_variable`, `width`

or

`LISTW` `prompt$`, `numerical_variable`, `width`, `item1$`, `item2$`, `item3$`, *etc.*

or

`LISTW` `prompt$`, `numerical_variable`, `width`, `height`

or

`LISTW` `prompt$`, `numerical_variable`, `width`, `height`, `item1$`, `item2$`, `item3$`, *etc.*

Arguments:

`prompt$`: the prompt
`numerical_variable`: the variable associated with the list
`width`: the width of the list
`height`: the number of items the list can display without scrolling
`item1$`, `item2$`, `item3$`, *etc.*: items to be placed in the list

Use this command to add a list that requires the user to select an item.

`prompt$` is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character ("_") after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is

set to the number of the item selected when the user presses a button other than the Cancel button.

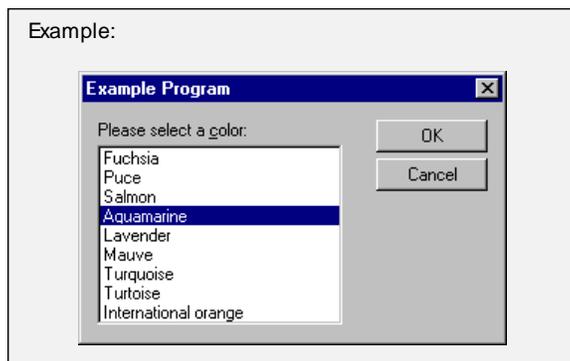
`width` is the minimum width of the list box. The list will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual list may be wider (but never narrower) than the width you specify, since lists always extend across the whole dialog. To calculate the width automatically, use the `LIST` command

`height` is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the `LISTW` command, or you can add items later using the `LITEM` command.

`LISTW` will disable all buttons added with the `BUTTON` command if no item is selected. To allow the user to proceed without selecting an item, use the `LISTW0` command.

If you want the items in the list to be sorted alphabetically, use the `SLISTW` command.



If the value of `numerical_variable` is negative, or if it contains fractions, `LISTW` generates an error.

If `width` or `height` is negative or zero, or if it contains fractions, `LISTW` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `LIST0` Command

Adds an unsorted auto-width list to the current dialog. The list does not require a user selection.

Syntax:

LIST0 `prompt$,numerical_variable`

or

LIST0 `prompt$,numerical_variable,item1$,item2$,item3$, etc.`

or

LIST0 `prompt$,numerical_variable,height`

or

LIST0 *prompt\$,numerical_variable,height,item1\$,item2\$,item3\$, etc.*

Arguments:

prompt\$: the prompt
numerical_variable: the variable associated with the list
height: the number of items the list can display without scrolling
item1\$,item2\$,item3\$, etc.: items to be placed in the list

Use this command to add a list that does not require the user to select an item.

prompt\$ is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character ("_") after the appropriate character in the prompt.

numerical_variable has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button. If no item is selected, the value is set to 0.

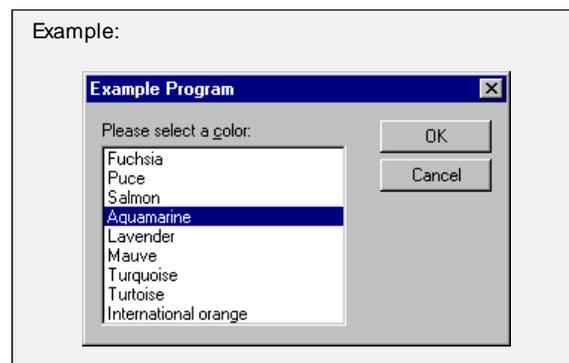
The list will be at least wide enough to hold the longest of the items you add. To specify a specific width for the list, use the **LISTW0** command.

height is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the **LIST0** command, or you can add items later using the **LITEM** command.

LIST0 does not require the user make a selection. If you want to force the user to select an item, use the **LIST** command.

If you want the items in the list to be sorted alphabetically, use the **SLIST0** command.



If the value of **numerical_variable** is negative, or if it contains fractions, **LIST0** generates an error.

If **height** is negative or zero, or if it contains fractions, **LIST0** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **LISTWO** Command

Adds an unsorted fixed-width list to the current dialog. The list does not require a user selection.

Syntax:

LISTWO *prompt\$,numerical_variable,width*

or

LISTWO *prompt\$,numerical_variable,width,item1\$,item2\$,item3\$, etc.*

or

LISTWO *prompt\$,numerical_variable,width,height*

or

LISTWO *prompt\$,numerical_variable,width,height,item1\$,item2\$,item3\$, etc.*

Arguments:

<i>prompt\$:</i>	the prompt
<i>numerical_variable:</i>	the variable associated with the list
<i>width:</i>	the width of the list
<i>height:</i>	the number of items the list can display without scrolling
<i>item1\$,item2\$,item3\$, etc.:</i>	items to be placed in the list

Use this command to add a list that does not require the user to select an item.

prompt\$ is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character ("_") after the appropriate character in the prompt.

numerical_variable has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button. If no item is selected, the value is set to 0.

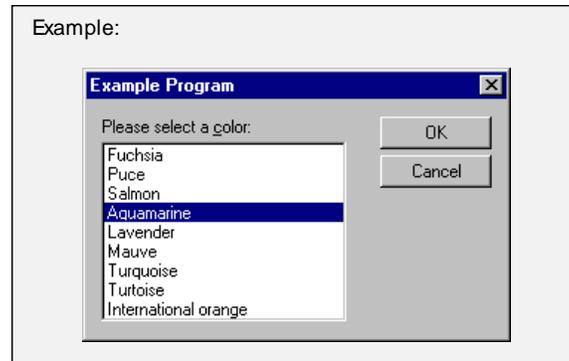
width is the minimum width of the list box. The list will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual list may be wider (but never narrower) than the width you specify, since lists always extend across the whole dialog. To calculate the width automatically, use the **LIST0** command

height is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the **LISTWO** command, or you can add items later using the **LITEM** command.

LISTW0 does not require the user make a selection. If you want to force the user to select an item, use the LISTW command.

If you want the items in the list to be sorted alphabetically, use the SLISTW0 command.



If the value of numerical_variable is negative, or if it contains fractions, LISTW0 generates an error.

If width or height is negative or zero, or if it contains fractions, LISTW0 generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The SLIST Command

Adds a sorted auto-width list to the current dialog. The list requires a user selection.

Syntax:

SLIST prompt\$,numerical_variable

or

SLIST prompt\$,numerical_variable,item1\$,item2\$,item3\$, etc.

or

SLIST prompt\$,numerical_variable,height

or

SLIST prompt\$,numerical_variable,height,item1\$,item2\$,item3\$, etc.

Arguments:

prompt\$: the prompt

numerical_variable: the variable associated with the list

height: the number of items the list can display without scrolling

item1\$,item2\$,item3\$, etc.: items to be placed in the list

Use this command to add a list that requires the user to select an item. The items will be sorted alphabetically.

prompt\$ is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character (“_”) after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button.

Note that the items are numbered according to the order in which you add them, not the order in which they appear in the list. If `numerical_variable` is one, `item1$` will be selected initially. Since the list was sorted, however, `item1$` might not be the first item in the list in the dialog. If the user selects `item2$`, `numerical_variable` will be set to two, regardless of where `item2$` appears in the sorted list.

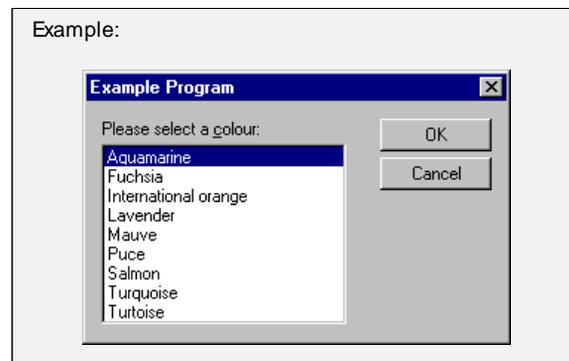
The list will be at least wide enough to hold the longest of the items you add. To specify a specific width for the list, use the `SLISTW` command.

`height` is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the `SLIST` command, or you can add items later using the `LITEM` command.

`SLIST` will disable all buttons added with the `BUTTON` command if no item is selected. To allow the user to proceed without selecting an item, use the `SLIST0` command.

`SLIST` sorts all items alphabetically. If you want the items in the list to appear in the order you add them, use the `LIST` command.



If the value of `numerical_variable` is negative, or if it contains fractions, `SLIST` generates an error.

If `height` is negative or zero, or if it contains fractions, `SLIST` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `SLISTW` Command

Adds a sorted fixed-width list to the current dialog. The list requires a user selection.

Syntax:

SLISTW prompt\$, numerical_variable, width

or

SLISTW *prompt\$,numerical_variable,width,item1\$,item2\$,item3\$, etc.*

or

SLISTW *prompt\$,numerical_variable,width,height*

or

SLISTW *prompt\$,numerical_variable,width,height,item1\$,item2\$,item3\$, etc.*

Arguments:

prompt\$: the prompt
numerical_variable: the variable associated with the list
width: the width of the list
height: the number of items the list can display without scrolling
item1\$,item2\$,item3\$, etc.: items to be placed in the list

Use this command to add a list that requires the user to select an item. The items will be sorted alphabetically.

prompt\$ is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character (“_”) after the appropriate character in the prompt.

numerical_variable has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button.

Note that the items are numbered according to the order in which you add them, not the order in which they appear in the list. If **numerical_variable** is one, **item1\$** will be selected initially. Since the list was sorted, however, **item1\$** might not be the first item in the list in the dialog. If the user selects **item2\$**, **numerical_variable** will be set to two, regardless of where **item2\$** appears in the sorted list.

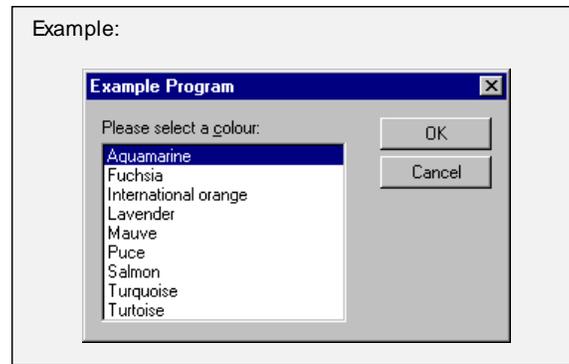
width is the minimum width of the list box. The list will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual list may be wider (but never narrower) than the width you specify, since lists always extend across the whole dialog. To calculate the width automatically, use the **SLIST** command

height is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the **SLISTW** command, or you can add items later using the **LITEM** command.

SLISTW will disable all buttons added with the **BUTTON** command if no item is selected. To allow the user to proceed without selecting an item, use the **SLISTW0** command.

SLISTW sorts all items alphabetically. If you want the items in the list to appear in the order you add them, use the **LISTW** command.



If the value of `numerical_variable` is negative, or if it contains fractions, `SLISTW` generates an error.

If width or height is negative or zero, or if it contains fractions, `SLISTW` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `SLIST0` Command

Adds a sorted auto-width list to the current dialog. The list does not require a user selection.

Syntax:

SLIST0 `prompt$,numerical_variable`

or

SLIST0 `prompt$,numerical_variable,item1$,item2$,item3$, etc.`

or

SLIST0 `prompt$,numerical_variable,height`

or

SLIST0 `prompt$,numerical_variable,height,item1$,item2$,item3$, etc.`

Arguments:

`prompt$:` the prompt

`numerical_variable:` the variable associated with the list

`height:` the number of items the list can display without scrolling

`item1$,item2$,item3$, etc.:` items to be placed in the list

Use this command to add a list that does not require the user to select an item. The items will be sorted alphabetically.

`prompt$` is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character (“_”) after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button. If no item is selected, the value is set to 0.

Note that the items are numbered according to the order in which you add them, not the order in which they appear in the list. If `numerical_variable` is one, `item1$` will be selected initially. Since the list was sorted, however, `item1$` might not be the first item in the list in the dialog. If the user selects `item2$`, `numerical_variable` will be set to two, regardless of where `item2$` appears in the sorted list.

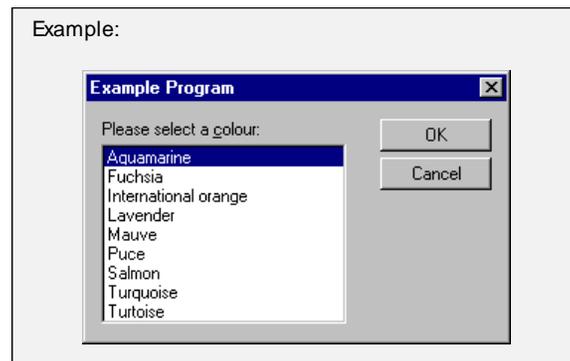
The list will be at least wide enough to hold the longest of the items you add. To specify a specific width for the list, use the, use the `SLISTW0` command.

`height` is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the `SLIST0` command, or you can add items later using the `LITEM` command.

`SLIST0` does not require the user make a selection. If you want to force the user to select an item, use the `SLIST` command.

`SLIST0` sorts all items alphabetically. If you want the items in the list to appear in the order you add them, use the `LIST0` command.



If the value of `numerical_variable` is negative, or if it contains fractions, `SLIST0` generates an error.

If `height` is negative or zero, or if it contains fractions, `SLIST0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `SLISTW0` Command

Adds a sorted fixed-width list to the current dialog. The list does not require a user selection.

Syntax:

SLISTW0 `prompt$,numerical_variable,width`

or

SLISTW0 `prompt$,numerical_variable,width,item1$,item2$,item3$, etc.`

or

SLISTW0 `prompt$,numerical_variable,width,height`

or

SLISTW0 `prompt$,numerical_variable,width,height,item1$,item2$,item3$, etc.`

Arguments:

`prompt$:` the prompt
`numerical_variable:` the variable associated with the list
`width:` the width of the list
`height:` the number of items the list can display without scrolling
`item1$,item2$,item3$, etc.:` items to be placed in the list

Use this command to add a list that does not require the user to select an item. The items will be sorted alphabetically.

`prompt$` is a string that tells the user what the selection in the list is for. The prompt is placed immediately above the list. Use an empty string as the prompt if you do not want a prompt.

To assign a keyboard shortcut to the list, put an underscore character (“_”) after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button. If no item is selected, the value is set to 0.

Note that the items are numbered according to the order in which you add them, not the order in which they appear in the list. If `numerical_variable` is one, `item1$` will be selected initially. Since the list was sorted, however, `item1$` might not be the first item in the list in the dialog. If the user selects `item2$`, `numerical_variable` will be set to two, regardless of where `item2$` appears in the sorted list.

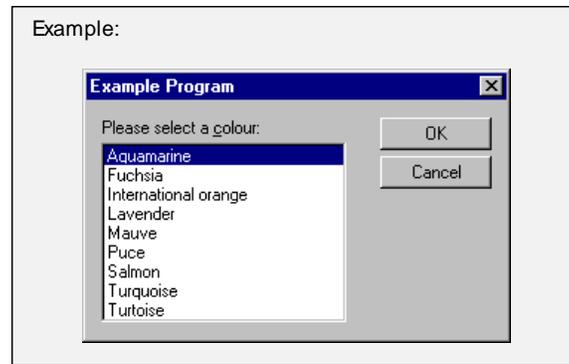
`width` is the minimum width of the list box. The list will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual list may be wider (but never narrower) than the width you specify, since lists always extend across the whole dialog. To calculate the width automatically, use the `SLIST0` command.

`height` is an optional height of the edit field. If you specify a height, the list will be able to show that many items at a time. The user can see additional items by scrolling down. If you don't specify a height, the list will be large enough to hold all the items.

You can specify any number of items to be placed in the list in the `SLISTW0` command, or you can add items later using the `LITEM` command.

`SLISTW0` does not require the user make a selection. If you want to force the user to select an item, use the `SLISTW` command.

`SLISTW0` sorts all items alphabetically. If you want the items in the list to appear in the order you add them, use the `LISTW0` command.



If the value of `numerical_variable` is negative, or if it contains fractions, `SLISTW0` generates an error.

If `width` or `height` is negative or zero, or if it contains fractions, `SLISTW0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **LITEM** Command

Adds items to a dialog list.

Syntax:

LITEM `item1$,item2$,item3$, etc.`

Arguments:

`item1$,item2$,item3$, etc.:` the items to be placed in the list

Use this command to add items to a list you just added to the current dialog using the `LIST`, `LISTW`, `LIST0`, `LISTW0`, `SLIST`, `SLISTW`, `SLIST0`, or `SLISTW0` command.

If the last item added to the current dialog was not a list, `LITEM` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **RDGRP** Command

Adds a radio button group to the current dialog. The user is required to select a button.

Syntax:

RDGRP `numerical_variable`

or

RDGRP `numerical_variable,label1$,label2$,label3$, etc.`

Arguments:

`numerical_variable:` the variable associated with the group

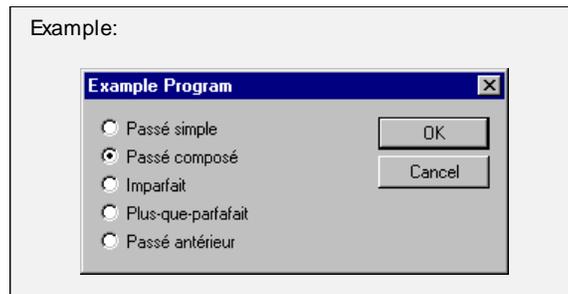
`label1$,label2$,label3$ etc.:` buttons to be added to the group

Use this command to add a radio group if the user must select one of the buttons.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the button with the corresponding number is selected initially. The variable value is set to the number of the button selected when the user presses a button other than the Cancel button.

You can specify any number of buttons to be added to the group in the `RDGRP` command, or you can add buttons later using the `RDBTN` command. To assign a keyboard shortcut to a button, put an underscore character ("_") after the appropriate character in its label.

`RDGRP` requires the user to select a button. If you want the user to be able to leave all buttons unclicked, use the `RDGRP0` command



If the value of `numerical_variable` is negative, or if it contains fractions, `RDGRP` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `RDGRP0` Command

Adds a radio button group to the current dialog. The user does not have to select a button.

Syntax:

RDGRP0 `numerical_variable`

or

RDGRP0 `numerical_variable, label1$, label2$, label3$ etc.`

Arguments:

`numerical_variable`: the variable associated with the group

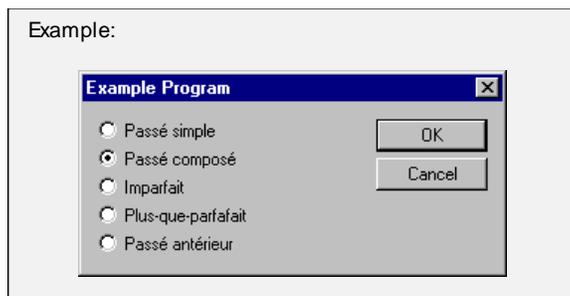
`label1$, label2$, label3$ etc.`: buttons to be added to the group

Use this command to add a radio group if the user does not have to select one of the buttons.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the button with the corresponding number is selected initially. The variable value is set to the number of the button selected when the user presses a button other than the Cancel button. If no button is clicked, the value is set to 0.

You can specify any number of buttons to be added to the group in the `RDGRP0` command, or you can add buttons later using the `RDBTN` command. To assign a keyboard shortcut to a button, put an underscore character ("_") after the appropriate character in its label.

`RDGRP` does not require the user to select a button. If you want to force the user to select a button, use the `RDGRP` command



If the value of `numerical_variable` is negative, or if it contains fractions, `RDGRP0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `RDBTN` Command

Adds buttons to a radio group.

Syntax:

RDBTN label1\$,label2\$,label3\$ *etc.*

Arguments:

label1\$,label2\$,label3\$ *etc.*: the buttons to be added to the group

Use this command to add buttons to a radio group you just added to the current dialog using the `RDGRP` or `RDGRP0` command. To assign a keyboard shortcut to a button, put an underscore character ("_") after the appropriate character in its label.

If the last item added to the current dialog was not a radio group, `RDBTN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `MENU` Command

Adds an auto-width pop-up menu to the current dialog. The menu requires a selection.

Syntax:

MENU prompt\$,numerical_variable

or

MENU prompt\$,numerical_variable,item1\$,item2\$,item3\$, *etc.*

or

MENU `prompt$,numerical_variable,width`

or

MENU `prompt$,numerical_variable,width,item1$,item2$,item3$, etc.`

Arguments:

`prompt$`: the prompt
`numerical_variable`: the variable associated with the pop-up menu
`width`: the width of the menu
`item1$,item2$,item3$, etc.`: items to be added to the menu

Use this command to add pop-up menu that requires the use to select an item. A pop-up menu is sometimes also called a combo-box.

`prompt$` is a string that tells the user what it is he is selecting. You can have three types of prompts:

- To have no prompt, use an empty string for `prompt$`.
- To have the prompt on a separate line, use a string that end with a return character (`RET$`)
- To have the prompt on the same line and to the left of the menu, use a string that does not end in a return character

To assign a keyboard shortcut to the pop-up menu, put an underscore character ("_") after the appropriate character in the prompt.

`numerical_variable` has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button.

`width` is an optional minimum width of the menu. If you specify a width, the menu will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual menu may be wider (but never narrower) than the width you specify, since menus always extend across the whole dialog. If you don't specify a width, the menu will be at least wide enough to hold the longest of the items you add.

You can specify any number of items to be placed in the pop-up menu in the **MENU** command, or you can add items later using the **MITEM** command.

MENU will disable all buttons added with the **BUTTON** command if no item is selected. To allow the user to proceed without selecting an item, use the **MENU0** command.

If the value of `numerical_variable` is negative, or if it contains fractions, **MENU** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **MENU0** Command

Adds an auto-width pop-up menu to the current dialog. The menu does not require a selection.

Syntax:

MENU0 *prompt\$,numerical_variable*

or

MENU0 *prompt\$,numerical_variable,item1\$,item2\$,item3\$, etc.*

or

MENU0 *prompt\$,numerical_variable,width*

or

MENU0 *prompt\$,numerical_variable,width,item1\$,item2\$,item3\$, etc.*

Arguments:

prompt\$: the prompt

numerical_variable: the variable associated with the pop-up menu

width: the width of the menu

item1\$,item2\$,item3\$, etc.: items to be added to the menu

Use this command to add pop-up menu that enables (but does not require)requires the use to select an item. A pop-up menu is sometimes also called a combo-box.

prompt\$ is a string that tells the user what it is he is selecting. You can have three types of prompts:

- To have no prompt, use an empty string for *prompt\$*.
- To have the prompt on a separate line, use a string that end with a return character (*RET\$*)
- To have the prompt on the same line and to the left of the menu, use a string that dous not end in a return character

To assign a keyboard shortcut to the pop-up menu, put an underscore character ("_") after the appropriate character in the prompt.

numerical_variable has to be a numerical variable. If the variable value is greater than 0, the item with the corresponding number is selected initially. The variable value is set to the number of the item selected when the user presses a button other than the Cancel button. If no item is selected, the value is set to 0.

width is an optional minimum width of the menu. If you specify a width, the menu will be wide enough for items with about that many characters. Please note that since not all characters are the same width, how many characters will fit in the list depends greatly on what characters they are. Note also that the actual menu may be wider (but never narrower) than the width you specify, since menus always extend across the whole dialog. If you don't specify a width, the menu will be at least wide enough to hold the longest of the items you add.

You can specify any number of items to be placed in the pop-up menu in the `MENU0` command, or you can add items later using the `MITEM` command.

`MENU0` does not require the user make a selection. If you want to force the user to select an item, use the `MENU` command.

If the value of `numerical_variable` is negative, or if it contains fractions, `MENU0` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `MITEM` Command

Adds items to a dialog menu.

Syntax:

MITEM item1\$,item2\$,item3\$, etc.

Arguments:

item1\$,item2\$,item3\$, etc.: the items to be added to the menu

Use this command to add items to a pop-up menu you just added to the current dialog using the `MENU` or `MENU0` command.

If the last item added to the current dialog was not a pop-up menu, `MITEM` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `BUTTON` Command

Adds a button to the current dialog.

Syntax:

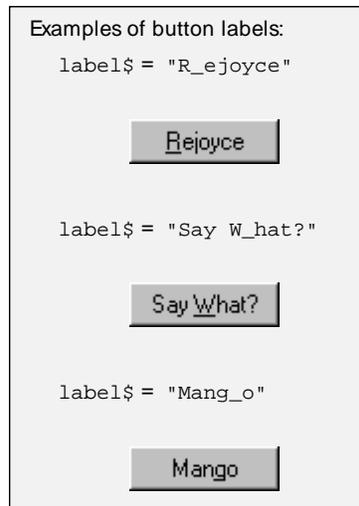
BUTTON label\$

Arguments:

label\$: the label on the button

This command adds a button to the dialog. The buttons added using the `BUTTON` and `BUTTON0` commands are numbered in the order you add them, starting at button number 1. See Dialog Buttons in *Dialogs* on page 29 for more details.

To assign a keyboard shortcut to the button, put an underscore character (" _ ") after the appropriate character in `label$`.



If you want your dialog to have only an OK and a Cancel button, you do not need to use the `BUTTON` command. If you do not add any buttons manually, SCL will add a standard OK button with button number 1, and a Cancel button with button number 0.

If you add any buttons using the `BUTTON`, `BUTTON0`, or `CANCELBTN` commands, you must add the OK and Cancel buttons manually.

The buttons added with the `BUTTON` command are disabled under the following conditions:

any entry field created using the `STREDIT`, `PWDEDIT`, `NUMEDIT`, or `INTEDIT` command has no text in it, or

any list box created using the `LIST`, `LISTW`, `SLIST`, or `SLISTW` command has no selection, or

any radio group created using the `RDGRP` command has no button pressed, or

any menu created using the `MENU` command has no selection.

To add a button that is always enabled, use the `BUTTON0` command.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `BUTTON0` Command

Adds a button that is always enabled to the current dialog.

Syntax:

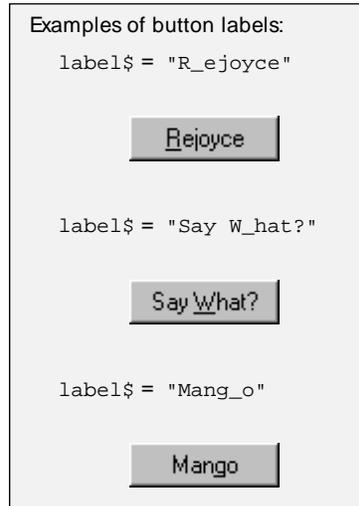
BUTTON0 label\$

Arguments:

label\$: the label on the button

This command adds a button to the dialog. The buttons added using the `BUTTON` and `BUTTON0` commands are numbered in the order you add them, starting at button number 1. See Dialog Buttons in *Dialogs* on page 29 for more details.

To assign a keyboard shortcut to the button, put an underscore character (“_”) after the appropriate character in `label$`.



If you want your dialog to have only an OK and a Cancel button, you do not need to use the `BUTTON0` command. If you do not add any buttons manually, SCL will add a standard OK button with button number 1, and a Cancel button with button number 0.

If you add any buttons using the `BUTTON`, `BUTTON0`, or `CANCELBTN` commands, you must add the OK and Cancel buttons manually.

The buttons added with the `BUTTON0` command are always enabled. To add a button that is automatically disabled if the user did not enter all required information, use the `BUTTON` command.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `CANCELBTN` Command

Adds a cancel button to the current dialog.

Syntax:

`CANCELBTN`

or

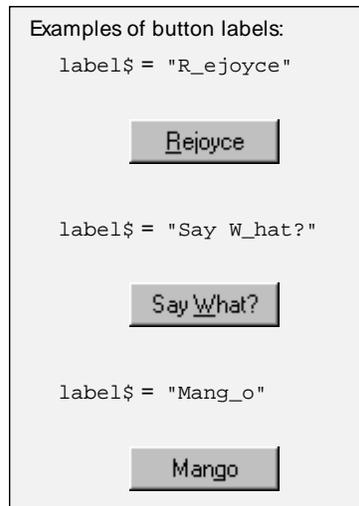
`CANCELBTN` `label$`

Arguments:

`label$`: the label on the button

This command adds a Cancel button to the dialog. The Cancel button is always button number 0. If you do not specify a label for the button, the button will read "Cancel."

To assign a keyboard shortcut to the button, put an underscore character (“_”) after the appropriate character in `label$`.



If you want your dialog to have only an OK and a Cancel button, you do not need to use the `CANCELBTN` command. If you do not add any buttons manually, SCL will add a standard OK button with button number 1, and a Cancel button with button number 0.

If you add any buttons using the `BUTTON`, `BUTTON0`, or `CANCELBTN` commands, you must add the OK and Cancel buttons manually.

If the current dialog already has a cancel button, `CANCELBTN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

Dialog Button Callback Commands

■ The `DLGERROR` Command

Highlights an item in a dialog to show the user where invalid data was entered.

Syntax:

DLGERROR item_variable

or

DLGERROR item_variable\$

or

DLGERROR item_variable%

Arguments:

item_variable, item_variable\$, item_variable%: the variable associated with the dialog item

If, inside a dialog button callback, you find that the user has entered invalid data, use the `PRINT` command to explain to the user what is wrong, then set the result variable specified in the `DIALOG` command to 0, and use the `DLGERROR` command to highlight the item where invalid data was entered.

If it is used outside a dialog button callback, or if `item_variable`, `item_variable$`, or `item_variable%` is not associated with a dialog item, `DLGERROR` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **SETLITEM** Command

Sets the text of a dialog list item.

Syntax:

SETLITEM `item_variable`,`index`,`item_text$`

Arguments:

`item_variable`: the variable associated with the list
`index`: the 1-based index of the list item
`item_text$`: the new list item text

Use `SETLITEM` inside a dialog button callback to change the text shown in a list item, or to restore a list item you deleted using the `DELLITEM` command..

Use `MAXLITEM` to determine the maximum list item index.

If you want to add an entirely new item at then (thereby increasing the maximum index by 1), use the `ADDLITEM` command.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a list, `SETLITEM` generates an error.

If `index` is smaller than 1, or larger than the maximum list item index, `SETLITEM` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **ADDLITEM** Command

Adds a new list item at the end of a dialog list.

Syntax:

ADDLITEM `item_variable`,`item_text$`

or

ADDLITEM `item_variable`,`item_text$`,`index_variable`

Arguments:

`item_variable`: the variable associated with the list
`item_text$`: the new list item text
`index_variable`: a variable to receive the index of the new list item

Use `ADDLITEM` inside a dialog button callback to add a new list item to a dialog list (thereby increasing the maximum index by 1). You can specify a variable that will be set to the index of the newly added item.

To change the text shown in an existing list item, or to restore a list item you deleted using the `DELLITEM` command, use the `SETLITEM` command.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a list, `SETLITEM` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The `DELLITEM` Command

Deletes a list item from a dialog list.

Syntax:

DELLITEM `item_variable`, `index`

or

DELLITEM `item_variable`, `index`, `next_index_variable`

Arguments:

`item_variable`: the variable associated with the list

`index`: the 1-based index of the list item

`next_index_variable`: a variable to receive the index of the next existing list item

Use `DELLITEM` inside a dialog button callback to delete a list item from a dialog list. Deleting a list item does not affect the indices of any other list items. The maximum item index is *not* changed. The item is merely removed from the list box in the dialog.

You can specify a variable to receive the index of the next item after the one you deleted (or before it, if there is none after). If you delete the last item in the list, the variable will be set to 0. You can use this feature if you are deleting the currently selected item. If you specify the item variable as the next index variable, then the next item in the list will automatically be selected. To delete the selected item of a list with the variable `list_variable`, use the following code:

```
DELLITEM list_variable, list_variable, list_variable
```

This will ensure that a new list item is selected, if possible. Please note that the item returned in `next_index_variable` is the next item *in the order in which the items appear in the list*, not the item with the next higher index.

Use `MAXLITEM` to determine the maximum list item index. Use `LITEMEXISTS%` to determine if the list item has already been deleted.

To delete all list items and rebuild the list from scratch, use the `CLRLITEMS` command.

You can restore an item that you deleted using the `SETLITEM` command.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a list, `DELLITEM` generates an error.

If `index` is smaller than 1, or larger than maximum list item index, or if the list item with the specified index has already been deleted, `DELLITEM` generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **CLRLITEMS** Command

Deletes all items in a dialog list.

Syntax:

CLRLITEMS *item_variable*

Arguments:

item_variable: the variable associated with the list

Use **CLRLITEMS** inside a dialog button callback to delete all items from a dialog list. If you use **CLRLITEMS**, the maximum index will be 0 (meaning no items exist), and the next item you add using **ADDLITEM** will have the item number 1. This distinguishes it from **DELLITEM**, which does not change the maximum item number or the indices of any items.

Use **DELLITEM** to remove a single item from a dialog list.

You *cannot* restore the items using the **SETLITEM** command after clearing the items. Use **ADDLITEM** to add new items instead.

If it is used outside a dialog button callback, or if *item_variable* is not associated with a list, **CLRLITEMS** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **SETMITEM** Command

Sets the text of a dialog menu item.

Syntax:

SETMITEM *item_variable,index,item_text\$*

Arguments:

item_variable: the variable associated with the menu

index: the 1-based index of the menu item

item_text\$: the new menu item text

Use **SETMITEM** inside a dialog button callback to change the text shown in a menu item, or to restore a menu item you deleted using the **DELMITEM** command..

Use **MAXMITEM** to determine the maximum menu item index.

If you want to add an entirely new item at then (thereby increasing the maximum index by 1), use the **ADDMITEM** command.

If it is used outside a dialog button callback, or if *item_variable* is not associated with a menu, **SETMITEM** generates an error.

If *index* is smaller than 1, or larger than the maximum menu item index, **SETMITEM** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **ADDMITEM** Command

Adds a new menu item at the end of a dialog menu.

Syntax:

ADDMITEM *item_variable,item_text*\$

or

ADDMITEM *item_variable,item_text*\$,*index_variable*

Arguments:

item_variable: the variable associated with the menu

item_text\$: the new menu item text

index_variable: a variable to receive the index of the new menu item

Use **ADDMITEM** inside a dialog button callback to add a new menu item to a dialog menu (thereby increasing the maximum index by 1). You can specify a variable that will be set to the index of the newly added item.

To change the text shown in an existing menu item, or to restore a menu item you deleted using the **DELMITEM** command, use the **SETMITEM** command.

If it is used outside a dialog button callback, or if *item_variable* is not associated with a menu, **SETMITEM** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **DELMITEM** Command

Deletes a menu item from a dialog menu.

Syntax:

DELMITEM *item_variable,index*

or

DELMITEM *item_variable,index,next_index_variable*

Arguments:

item_variable: the variable associated with the menu

index: the 1-based index of the menu item

next_index_variable: a variable to receive the index of the next existing menu item

Use **DELMITEM** inside a dialog button callback to delete a menu item from a dialog menu. Deleting a menu item does not affect the indices of any other menu items. The maximum item index is *not* changed. The item is merely removed from the menu box in the dialog.

You can specify a variable to receive the index of the next item after the one you deleted (or before it, if there is none after). If you delete the last item in the menu, the variable will be set to 0. You can use this feature if you are deleting the currently selected item. If you specify the item variable as the next index variable, then the next item in the menu will automatically be selected. To delete the selected item of a menu with the variable *menu_variable*, use the following code:

DELMITEM menu_variable,menu_variable,menu_variable

This will ensure that a new menu item is selected, if possible.

Use **MAXMITEM** to determine the maximum menu item index. Use **MITEMEXISTS%** to determine if the menu item has already been deleted.

To delete all menu items and rebuild the menu from scratch, use the **CLRMITEMS** command.

You can restore an item that you deleted using the **SETMITEM** command.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, **DELMITEM** generates an error.

If `index` is smaller than 1, or larger than maximum menu item index, or if the menu item with the specified index has already been deleted, **DELMITEM** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

■ The **CLRMITEMS** Command

Deletes all items in a dialog menu.

Syntax:

CLRMITEMS item_variable

Arguments:

`item_variable`: the variable associated with the menu

Use **CLRMITEMS** inside a dialog button callback to delete all items from a dialog menu. If you use **CLRMITEMS**, the maximum index will be 0 (meaning no items exist), and the next item you add using **ADDMITEM** will have the item number 1. This distinguishes it from **DELMITEM**, which does not change the maximum item number or the indices of any items.

Use **DELMITEM** to remove a single item from a dialog menu.

You *cannot* restore the items using the **SETMITEM** command after clearing the items. Use **ADDMITEM** to add new items instead.

If it is used outside a dialog button callback, or if `item_variable` is not associated with a menu, **CLRMITEMS** generates an error.

This command is not available within programs for sources, checksums, and SABus response data, programs for SABus commands, or RTS controls.

File and Network Connection Commands

■ The **OPEN** Command

Opens a file and assigns a file number to it.

Syntax:

OPEN file_number,path\$,access_mode,file_type

or

OPEN file_number,path\$,access_mode,file_type,result_variable%

Arguments:

file_number: the file number to assign to the file (0 to 4,294,967,295)

path\$: the path of the file

access_mode: the access mode (see below)

file_type: the file type (see below)

result_variable%: a variable to receive the success/failure status

access_mode can be one of the following values:

Value	mode	meaning
1	read	open at beginning for reading
2	write	delete file content and open for writing
3	ap- pend	open at end for reading and writing

file_type can be one of the following values:

Value	type	meaning
1	text file	translate CR+LF pairs
2	binary file	don't translate CR+LF pairs

If the file cannot be opened for any reason, **OPEN** will display a message to the user. If you do not specify a result variable, **OPEN** will end the program if the file cannot be opened. If you do specify a result variable, it will be set to true if the file could be opened, and false if it could not.

If file_number is not a valid file number, or if any open file or network connection is already using it, **OPEN** generates an error.

If access_mode or file_type is not one of the values listed above, **OPEN** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **CONNECT** Command

Opens a TCP/IP socket network connection and assigns a file number to it.

Syntax:

CONNECT file_number,ip_address\$,tcp_port_number

or

CONNECT file_number,ip_address\$,tcp_port_number,result_variable%

or

CONNECT

`file_number, ip_address$, tcp_port_number, result_variable%, timeout`

Arguments:

`file_number`: the file number to assign to the file (0 to 4,294,967,295)
`ip_address$`: the IP address of the server
`tcp_port_number`: the TCP port number that the server is waiting on
`result_variable%`: a variable to receive the success/failure status
`timeout`: the timeout in milliseconds for this connection for the `INPUT#` command

`CONNECT` attempts to establish a TCP/IP socket connection to a server waiting on the specified port at the specified IP address. `ip_address$` is the server's IP address in dotted form, e.g. "192.168.1.100". The first component of an IP address cannot be 0, 127, or greater than 223, except for the loopback address 127.0.0.1.

Unlike the `OPEN` command, `CONNECT` will *not* display an error message to the user if the connection could not be established. If you do not specify a result variable, `CONNECT` will end the program if the connection cannot be established. If you do specify a result variable, it will be set to true if the connection could be established, and false if it could not.

If `file_number` is not a valid file number, or if any open file or network connection is already using it, `CONNECT` generates an error.

If `ip_address` is not a valid IP address, or `tcp_port_number` is not a valid TCP port, `CONNECT` generates an error.

The `timeout` value sets the number of milliseconds for the connection. If this value is not given, the connection will have a default timeout of 250 ms. If the value is not an integer greater than 0, `CONNECT` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `CLOSE` Command

Closes an open file or network connection.

Syntax:

CLOSE `file_number`

Arguments:

`file_number`: the file number of the file or connection

If `file_number` does not represent an open file or network connection, `CLOSE` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **PRINT#** Command

Writes data to a file at the current file position, or sends data over a network connection.

Syntax:

PRINT# *file_number* , *list of expressions*

Arguments:

file_number: the file number of the file or connection

PRINT# is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are written to the file directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the file.

PRINT# will write a line termination (CR for binary files and network connections, CR+LF for text files) after the last expression unless you do one of the following:

To omit the line termination, put a semicolon (;) at the end of the list of expressions.

To replace the line termination by a tab character, put a comma at the end of the list of expressions.

If the file is a text file, **PRINT#** will translate CRs into CR+LF pairs.

If *file_number* does not represent a file opened for writing or appending or a network connection, **PRINT#** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **INPUT#** Command

Reads data from a file beginning at the current file position, or waits for data from a network connection.

Syntax:

INPUT# *file_number* , *data\$*

or

INPUT# *file_number* , *data\$* , *length*

or

INPUT# *file_number* , *data\$* , *length* , *tries*

Arguments:

file_number: the file number of the file or connection

data\$: a variable to receive the data

length: the number of characters to read

tries: the number of times the command will try to read from the connection

The behaviour of **INPUT#** differs slightly from files to network connections.

If `file_number` is a file:

If you specify a length, `INPUT#` will read that many characters, or all data to the end of the file, whichever is less. If you do not specify a length, `INPUT#` will read one line of text, including the line termination (CR for binary files, CR+LF for text files), or all data to the end of the file, whichever is less. If the file is a text file, `INPUT#` will translate CR+LF pairs into CRs.

If `file_number` is a network connection:

For network connections, you must specify a length. `INPUT#` will wait for data to come in over the network until that many bytes have been received, or the connection was closed by the remote computer. If no `tries` is given or equal to 0, U.P.M.A.C.S. will keep on trying to read until the requested number of bytes is read. If `tries` is given, the number determines the number of reads performed in order to get the requested number of bytes. The time of each read attempt is determined by the `timeout` value set in the `CONNECT` command. Once the number of bytes has been received or the number of tries has been exhausted, the function will return. `data$` will contain all bytes that have been accumulated in the repeated tries including the empty string if no bytes were received at all. `INPUT#` will also stop waiting for data if you close the station.

If `file_number` does not represent a file opened for reading or appending or a network connection, or if `length` is not an integer between 1 and 4,294,967,295, `INPUT#` generates an error.

If `file_number` represents a network connection and you did not specify a length, `INPUT#` generates an error.

If `tries` is not an integer equal to or greater than 0, `INPUT#` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETFPOS` Command

Sets the current file position.

Syntax:

SETFPOS `file_number`,`byte_offset`

Arguments:

`file_number`: the file number of the file

`byte_offset`: the new offset in bytes from the beginning

Sets the file position to the specified number of bytes after the first byte in the file. The next `PRINT#` or `INPUT#` command will start at that position. If `file_number` is a text file, the position in bytes is not necessarily the position in characters, since CRs are stored as a CR-LF combination in Windows. There is no way to set the position within a text file in characters. If you require to do so, you must keep track of the file position yourself.

If `file_number` does not represent an open file, or if `byte_offset` is not an integer between 0 and 4,294,967,295, `SETFPOS` generates an error.

If `file_number` does represents a network connection, `SETFPOS` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **LIMITFLEN** Command

Limits the length of a file.

Syntax:

LIMITFLEN file_number,maximum_length

Arguments:

file_number: the file number of the file

maximum_length: the maximum allowed length

Checks if the file is larger than the specified maximum, and shrinks it if it is. The file is shrunk by removing lines of text from the beginning of the file, until the file is shorter than half of maximum_length. The file must have been opened for appending (reading and writing).

Use this command to keep a file, like a log file, to which you continuously add lines of text from eventually filling up all available disk space. Use **LIMITFLEN** everytime you have finished writing an entry to shrink the file if it has become too large.

Note: **LIMITFLEN** only removes entire lines from the file. Don't use **LIMITFLEN** on a file that does not consist of lines of text: it will not work properly and may simply erase the file altogether if it is too large.

If file_number does not represent a file opened for appending, or if maximum_length is not an integer between 1 and 4,294,967,295, **LIMITFLEN** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Register Commands

■ The **SETREGNAME** Command

Sets the name of a register.

Syntax:

SETREGNAME tag\$,name\$

Arguments:

tag\$: the tag of the register

name\$: the new name

If the register is user configurable, the new name will be remembered even if the station file is reloaded. If the register is not user configurable, the new name will be lost when the station is closed.

If tag\$ is not the tag of a register, **SETREGNAME** generates an error.

■ The **REVERTREGNAME** Command

Reverts the name of a register to the name defined in the station file.

Syntax:

REVERTREGNAME tag\$

Arguments:

tag\$: the tag of the register

Reverts a register to its original name, removing any names set using the **SETREGNAME** command, as well as any names configured by the user from within the Configure Data dialog.

If tag\$ is not the tag of a register, **REVERTREGNAME** generates an error.

■ The **SETONLOGSTR** Command

Sets the log string for the ON state of a register.

Syntax:

SETONLOGSTR tag\$,log_string\$

Arguments:

tag\$: the tag of the register

log_string\$: the new log string

The log string is written to the log as is; %-symbols are not replaced with the register name as is done with the default log strings.

If the register is user configurable and has configurable log strings, the new string will be remembered even if the station file is reloaded. If the register's log strings are not user configurable, the new string will be lost when the station is closed.

If tag\$ is not the tag of a register, **SETONLOGSTR** generates an error.

■ The **REVERTONLOGSTR** Command

Reverts the log string for the ON state of a register to the string defined in the station file.

Syntax:

REVERTONLOGSTR tag\$

Arguments:

tag\$: the tag of the register

Reverts a register's ON state log string to the original string, removing any strings set using the **SETONLOGSTR** command, as well as any strings configured by the user from within the Configure Data dialog.

If tag\$ is not the tag of a register, **REVERTONLOGSTR** generates an error.

■ The **SETOFFLOGSTR** Command

Sets the log string for the OFF state of a register.

Syntax:

SETOFFLOGSTR tag\$,log_string\$

Arguments:

tag\$: the tag of the register

log_string\$: the new log string

The log string is written to the log as is; %-symbols are not replaced with the register name as is done with the default log strings.

If the register is user configurable and has configurable log strings, the new string will be remembered even if the station file is reloaded. If the register's log strings are not user configurable, the new string will be lost when the station is closed.

If tag\$ is not the tag of a register, SETOFFLOGSTR generates an error.

■ The **REVERTOFFLOGSTR** Command

Reverts the log string for the OFF state of a register to the string defined in the station file.

Syntax:

REVERTOFFLOGSTR tag\$

Arguments:

tag\$: the tag of the register

Reverts a register's OFF state log string to the original string, removing any strings set using the SETOFFLOGSTR command, as well as any strings configured by the user from within the Configure Data dialog.

If tag\$ is not the tag of a register, REVERTOFFLOGSTR generates an error.

■ The **SETBSTVAL** Command

Sets the value of a bistate register.

Syntax:

SETBSTVAL tag\$,value%

Arguments:

tag\$: the tag of the register

value%: the value to set

If value% is true, the register will go into its ON/alarm state. If value% is false, it will go into its OFF/alarm clear state. If the register has a response delay, it will not only go into the specified state after the delay has elapsed without further change.

If the register is not masked, and the value changes as a result of the SETBSTVAL command, the change will be logged, and any automatic controls will be executed.

If tag\$ is not the tag of a bistate register, SETBSTVAL generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETBSTDLY** Command

Sets the response delay of a bistate register.

Syntax:

SETBSTDLY tag\$,delay_seconds

Arguments:

tag\$: the tag of the register

delay_seconds: the new response delay, in seconds

The new response delay will be active the next time the value of the register is updated.

If tag\$ is not the tag of a bistate register, or if delay_seconds is not between 0 and 4,294,967.295, SETBSTDLY generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETDIGVAL** Command

Sets the value of a digital register.

Syntax:

SETDIGVAL tag\$,value

or

SETDIGVAL tag\$,value_name\$

Arguments:

tag\$: the tag of the register

value: the value to set

value_name\$: the name of the value to set

If the register is not masked, and the value changes as a result of the SETDIGVAL command, the change will be logged, the alarm state will be updated, and any automatic controls will be executed.

If tag\$ is not the tag of a digital register, if value is not an integer between 0 and 4,294,967,295, or if value_name\$ is not the name of a value of the register, SETDIGVAL generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETANAVAL** Command

Sets one value of an analog register.

Syntax:**SETANAVAL** tag\$,value*or***SETANAVAL** tag\$,value,indexArguments:

tag\$: the tag of the register

value: the value to set

index: the 1-based index of the value

SETANAVAL sets a single value of an analog register, using "normal" (equal) as the greater / less status. To set a value with a different greater / less status, use SETANAVALGL.

index is the index of the value. A register with a size of one value has only a value with index 1. A register with a size of 4 values has values with indices 1, 2, 3, and 4.

If the register has a size of one value, you do not have to specify an index, as it is always 1. If the register has a size of more than one value, and you do not specify an index, then all existing values will be shifted to the next lower index, and the value with the highest index will be set to value. This allows you to create a history of values by calling SETANAVAL repeatedly at regular intervals.

If the register was in the error state, *all* values will be set to value.

If the register is not masked, the alarm state will be updated based on the new value, and any changes will be logged and automatic controls executed, if necessary.

To set all the values of an analog register with a size of more than one value at once, use the SETANAVALS command.

If tag\$ is not the tag of an analog register, or if index is smaller than 1, larger than the size of the register, or if it contains fractions, SETANAVAL generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETANAVALGL** Command

Sets one value and its greater / less status of an analog register.

Syntax:

SETANAVALGL tag\$,value,greater_less_status

or

SETANAVALGL tag\$,value,greater_less_status,index

Arguments:

tag\$:	the tag of the register
value:	the value to set
greater_less_status:	the greater / less status of the value
index:	the 1-based index of the value

SETANAVALGL sets a single value of an analog register. The greater / less status of the value will be set to "less than" if `greater_less_status` is less than 0, and to "greater than" if it is greater than 0. If `greater_less_status` is equal to zero, the value will be a normal value (equal). To set the value to a normal value, you can also use the **SETANAVAL** command.

`index` is the index of the value. A register with a size of one value has only a value with index 1. A register with a size of 4 values has values with indices 1, 2, 3, and 4.

If the register has a size of one value, you do not have to specify an index, as it is always 1. If the register has a size of more than one value, and you do not specify an index, then all existing values will be shifted to the next lower index, and the value with the highest index will be set to `value`. This allows you to create a history of values by calling **SETANAVALGL** repeatedly at regular intervals.

If the register was in the error state, *all* values will be set to `value` and `greater_less_status`.

If the register is not masked, the alarm state will be updated based on the new value, and any changes will be logged and automatic controls executed, if necessary.

To set all the values and greater / less statuses of an analog register with a size of more than one value at once, use the `SETANAVALSGL` command.

If `tag$` is not the tag of an analog register, or if `index` is smaller than 1, larger than the size of the register, or if it contains fractions, `SETANAVALGL` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETANAVALS` Command

Sets all the values of an analog register at once.

Syntax:

SETANAVALS `tag$,value_array`

Arguments:

`tag$`: the tag of the register
`value_array`: an array containing the values to set

`SETANAVALS` sets all the values of an analog register at the same time, using “normal” (equal) as the greater / less status. To set values with different greater / less statuses, use `SETANAVALSGL`.

`value_array` must be a 1-dimensional array variable that contains the new values. `value_array[1]` contains the value with index 1, `value_array[2]` contains the value with index 2, etc.. The index of the first value is 1.

Do not specify an index for `value_array`. Only use the array name, without subscript.

If the register is not masked, the alarm state will be updated based on the new values, and any changes will be logged and automatic controls executed, if necessary.

To set a single value of an analog register, use the `SETANAVAL` command.

If `tag$` is not the tag of an analog register, or if `value_array` is not a 1-dimensional array, `SETANAVALS` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETANAVALSGL` Command

Sets all the values and greater / less status of an analog register at once.

Syntax:

SETANAVALSGL `tag$,value_array,greater_less_array`

Arguments:

`tag$`: the tag of the register
`value_array`: an array containing the values to set
`greater_less_array`: an array containing the greater / less status of the values

`SETANAVALS` sets all the values of an analog register at the same time.

`value_array` and `greater_less_array` must be a 1-dimensional array variable that contains the new values and their greater / less statuses. `value_array[1]` and `greater_less_array[1]` contain the information for the value with index 1, `value_array[2]` and `greater_less_array[2]` contain the information for the value with index 2, etc.. The index of the first value is 1.

`value_array` contains the actual values. `greater_less_array` contains the greater less flags. The greater / less status of a value will be set to "less than" if the corresponding element of `greater_less_array` is less than 0, and to "greater than" if it is greater than 0. If an element is equal to zero, the corresponding value will be a normal value (equal). To set all the values to normal values, you use the `SETANAVALS` command.

Do not specify an index for `value_array` or `greater_less_array`. Only use the array names, without subscript.

If the register is not masked, the alarm state will be updated based on the new values, and any changes will be logged and automatic controls executed, if necessary.

To set a single value and greater / less status of an analog register, use the `SETANAVALGL` command.

If `tag$` is not the tag of an analog register, or if `value_array` or `greater_less_array` is not a 1-dimensional array, `SETANAVALSGL` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETANAMIN` Command

Sets the low limit of an analog register.

Syntax:

SETANAMIN `tag$,limit`

Arguments:

`tag$`: the tag of the register

`limit`: the new limit to set

If the register is not masked, the alarm state will be updated based on the new limit, and any changes will be logged and automatic controls executed, if necessary.

If `tag$` is not the tag of an analog register, `SETANAMIN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `CLRANAMIN` Command

Removes the low limit of an analog register.

Syntax:

CLRANAMIN `tag$`

Arguments:

`tag$`: the tag of the register

If the register is not masked, the alarm state will be updated based on the new limit, and any changes will be logged and automatic controls executed, if necessary.

If `tag$` is not the tag of an analog register, `CLRANAMIN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETANAMAX** Command

Sets the high limit of an analog register.

Syntax:

SETANAMAX `tag$,limit`

Arguments:

`tag$`: the tag of the register

`limit`: the new limit to set

If the register is not masked, the alarm state will be updated based on the new limit, and any changes will be logged and automatic controls executed, if necessary.

If `tag$` is not the tag of an analog register, `SETANAMAX` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **CLRANAMAX** Command

Removes the high limit of an analog register.

Syntax:

CLRANAMAX `tag$`

Arguments:

`tag$`: the tag of the register

If the register is not masked, the alarm state will be updated based on the new limit, and any changes will be logged and automatic controls executed, if necessary.

If `tag$` is not the tag of an analog register, `CLRANAMAX` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETINDRANGE** Command

Sets the range of dials, graphs, and x-y markers of an analog register.

Syntax:

SETINDRANGE `tag$,lower_bound,upper_bound`

Arguments:

`tag$`: the tag of the register

`lower_bound`: the bottom or left bound of the indicators

`upper_bound`: the top or right bound of the indicators

`SETINDRANGE` sets the range of values displayed by all dials, graphs, and x-y position markers that display the value of a the analog register you specify. The indicators will be redrawn to reflect the new range.

You cannot set the ranges of individual indicators. `SETINDRANGE` sets the range of all indicators for the register to the same bounds. If you need to change the ranges of indicators individually, you must define separate registers for them.

To revert the indicators' range to the original bounds, use the `REVERTINDRANGE` command.

Do not confuse this command with the `SETANAMIN` and `SETANAMAX` commands, which set the alarm limits of the register.

If `tag$` is not the tag of an analog register `SETINDRANGE` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `REVERTINDRANGE` Command

Reverts the range of dials, graphs, and x-y markers of an analog register to their original bounds.

Syntax:

REVERTINDRANGE tag\$

Arguments:

tag\$: the tag of the register

`REVERTINDRANGE` reverts the range of values displayed by all dials, graphs, and x-y position markers that display the value of a the analog register you specify to their original values as specified in the station file. The indicators will be redrawn to reflect the new ranges.

To set all the indicators' range, use the `SETINDRANGE` command.

Do not confuse this command with the `CLRANAMIN` and `CLRANAMAX` commands, which clear the alarm limits of the register.

If `tag$` is not the tag of an analog register `REVERTINDRANGE` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETSTRVAL` Command

Sets the value of a string register.

Syntax:

SETSTRVAL tag\$,value\$

Arguments:

tag\$: the tag of the register

value\$: the value to set

If the register is not masked, the alarm state will be updated based on the new value, and any changes will be logged and automatic controls executed, if necessary.

If tag\$ is not the tag of a string register SETSTRVAL generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The MASK Command

Masks a register manually.

Syntax:

MASK tag\$

Arguments:

tag\$: the tag of the register

Masking a register using the MASK command is equivalent to selecting "Mask Data..." from the "Settings" menu and masking it from within the Mask/Unmask Data dialog. MASK is independent of auto masking and of internal masking. To mask a register internally, use the INTMASK command instead.

Use UNMASK to unmask the register.

If tag\$ is not the tag of a register, MASK generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The UNMASK Command

Unmasks a register manually.

Syntax:

UNMASK tag\$

Arguments:

tag\$: the tag of the register

Unmasking a register using the UNMASK command is equivalent to selecting "Mask Data..." from the "Settings" menu and unmasking it from within the Mask/Unmask Data dialog. UNMASK is independent of auto masking and of internal masking. To remove the internal mask from a register, use the INTMASK command instead.

If tag\$ is not the tag of a register, UNMASK generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **INTMASK** Command

Masks a register internally.

Syntax:

INTMASK tag\$

Arguments:

tag\$: the tag of the register

Masking a register using the **INTMASK** command applies a special internal mask to the register that cannot be removed by the operator. **INTMASK** is independent of manual masking and of automatic masking. To mask a register as if masked manually by the operator, use the **MASK** command instead.

Use **INTUNMASK** to remove the special internal mask.

If tag\$ is not the tag of a register, **INTMASK** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **INTUNMASK** Command

Unmasks a register internally.

Syntax:

INTUNMASK tag\$

Arguments:

tag\$: the tag of the register

Unmasking a register using the **INTUNMASK** command removes the special internal mask applied using the **INTMASK** command. **INTUNMASK** is independent of manual masking and of automatic masking. To unmask a register as if unmasked manually by the operator, use the **UNMASK** command instead.

If tag\$ is not the tag of a register, **INTUNMASK** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **HIDE** Command

Hides a register.

Syntax:

HIDE tag\$

Arguments:

tag\$: the tag of the register

Hiding a register removes all its indicators from the screen. It does not disable the register, or change its behaviour in any way. If you need to disable the register as well as hide it, use the `INTMASK` command in addition to the `HIDE` command.

If `tag$` is not the tag of a register, `HIDE` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `UNHIDE` Command

Unhides a hidden register.

Syntax:

UNHIDE tag\$

Arguments:

tag\$: the tag of the register

Unhides a register previously hidden using the `HIDE` command, or a register with the “initially hidden” flag.

If `tag$` is not the tag of a register, `UNHIDE` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Serial Communication Commands

■ The `GRAB` Command

Requests exclusive access to one or more serial ports.

Syntax:

GRAB tag1\$, tag2\$, tag3\$, etc.

Arguments:

tag1\$, tag2\$, tag3\$, etc.: the tags of the ports to request access to

Once you have grabbed a port, polling is interrupted, and no other program is allowed access to the port until this program has ended or used the `RELEASE` command.

If you have used the `GRAB` command, you will not be able to use it again until you have used the `RELEASE` command. This is to avoid deadlocks between SCL programs, each retaining exclusive access to one port while waiting for the one the other holds. For the same reason, you cannot use the `SEND CMD`, `SEND STR`, `SEND BIN`, `DISABLE DRV`, `ENABLE DRV`, `DISABLE CMD`, and `ENABLE CMD` commands on any other ports than the ones you have grabbed, if you have grabbed any.

You do not need to grab serial ports to access them. You only need to grab ports if you want to ensure that you will not be interrupted by a poll or another program between two commands that access the port.

In device driver programs, you must specify only one tag, and that tag must be an empty string (" "). This will grab the port of the device the program belongs to, the only port that can be accessed by a device driver program.

If any of the tags are not the tag of a serial port, or if any tag appears twice in the list, GRAB generates an error.

If you already have exclusive access to any serial ports when you invoke GRAB, it will generate an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **RELEASE** Command

Releases all ports to which the program has exclusive access.

Syntax:

RELEASE

If the program does not have exclusive access to any serial ports, RELEASE generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SENCMD** Command

Sends a command to a serial device and waits for a response, if necessary.

Syntax:

SENCMD port_tag\$, device_tag\$, command_tag\$, *command parameters*

Arguments:

port_tag\$: the tag of the serial port the command's device is attached to

device_tag\$: the tag of the device to send the command to

command_tag\$: the tag of the command

The tag of the command is followed by a list of parameters, separated by commas. You must specify an expression for each of the command's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

If you have requested exclusive access to any serial ports using the GRAB command, you cannot send commands to any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can send commands to any port you like.

SENCMD sends the command to the serial port even if the device driver has not been properly initialized. You can determine whether the device's initialization sequence has been sent successfully using the DRVREADY% function.

If the specified command or device driver has been disabled on the specified serial port, no data is sent.

If the command expects a response from the device, `SENDCMD` will wait for the response and update all data objects and registers that depend on it. You can use the `DRVSUCCESS%`, `DRVTIMNEOUT%`, and `DRVERROR%` reserved variables to see whether the response was received successfully. Use the `DRVDATA$`, `DRVERROR`, and `DRVERROR$` reserved variables or the `DRVNDATA$`, `DRVNERROR`, and `DRVNERROR$` functions to retrieve the data or error codes the device returned.

Note: If you specify a string variable after the tag of the command of a legacy device driver, `SENDCMD` will place the response data into that variable. This usage of `SENDCMD` is obsolete and should not be used. Use the `DRVDATA$` reserved variable to access the response data of a legacy device driver command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only commands of the device the program belongs to can be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `command_tag$` is not the tag of a command in the device's driver, `SENDCMD` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `SENDCMD` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SENDSTR` Command

Sends data to a serial port, appending the line termination and translating CRs.

Syntax:

SENDSTR port_tag\$,string\$

Arguments:

port_tag\$: the tag of the serial port

string\$: the string to send

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot send data to any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can send data to any port you like.

`SENDSTR` appends the line termination specified for the serial ports to the string, and converts any CRs in the string to the line termination. If you wish to send data without appending a line termination and without CR translation, use the `SENDBIN` command.

If `port_tag$` is not the tag of a serial port, `SENDSTR` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `SENDSTR` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Sending Custom Commands to Devices That Use Legacy Device Drivers

Legacy device drivers do not support parameters for commands. It is therefore often necessary to create commands on the fly. For this purpose, you can specify a response in a legacy device driver to be used for waiting for a response:

Syntax:

SENDSTR port_tag\$,string\$,device_tag\$,response_tag\$

or

SENDSTR port_tag\$,string\$,device_tag\$,response_tag\$,response_variable\$

Arguments:

device_tag\$: the tag of the device whose driver contains the response

response_tag\$: the tag of the response

response_variable\$: a variable that is to receive the device's response

If you specify a driver and response, **SENDSTR** will wait for a response using the response you specified, and all registers that are attached to it are automatically updated. If you also specify a response variable, the data is placed into that variable as well. Registers attached to the response are updated in either case. If the command timed out, `response_variable$` will be set to an empty string.

SENDSTR sends the data to the serial port even if the serial port has not been properly initialized. You can determine whether the device's initialization sequence has been sent successfully using the `DRVREADY%` function.

If `device_tag$` is not the tag of a device on the port that uses a legacy device driver, or `response_tag$` is not the tag of a response in that driver, **SENDSTR** generates an error.

■ The **SENCBIN** Command

Sends data to a serial port.

Syntax:

SENCBIN port_tag\$,data\$

Arguments:

port_tag\$: the tag of the serial port

data\$: the data to send

If you have requested exclusive access to any serial ports using the **GRAB** command, you cannot send data to any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can send data to any port you like.

SENCBIN does not append any line termination, nor does it do any CR translation. If you wish to send strings using the line termination specified in the serial ports, use the **SENDSTR** command.

If `port_tag$` is not the tag of a serial port, **SENCBIN** generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, **SENCBIN** generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Sending Custom Commands to Devices That Use Legacy Device Drivers

Legacy device drivers do not support parameters for commands. It is therefore often necessary to create commands on the fly. For this purpose, you can specify a response in a legacy device driver to be used for waiting for a response:

Syntax:

SENDERBIN port_tag\$,string\$,device_tag\$,response_tag\$

or

SENDERBIN port_tag\$,string\$,device_tag\$,response_tag\$,response_variable\$

Arguments:

device_tag\$: the tag of the device whose driver contains the response

response_tag\$: the tag of the response

response_variable\$: a variable that is to receive the device's response

If you specify a driver and response, **SENDERBIN** will wait for a response using the response you specified, and all registers that are attached to it are automatically updated. If you also specify a response variable, the data is placed into that variable as well. Registers attached to the response are updated in either case. If the command timed out, response_variable\$ will be set to an empty string.

SENDERBIN sends the data to the serial port even if the serial port has not been properly initialized. You can determine whether the device's initialization sequence has been sent successfully using the **DRVREADY%** function.

If device_tag\$ is not the tag of a device on the port that uses a legacy device driver, or response_tag\$ is not the tag of a response in that driver, **SENDERBIN** generates an error.

■ The **DISABLEDRV** Command

Disables a device on a serial port.

Syntax:

DISABLEDRV port_tag\$,device_tag\$

Arguments:

port_tag\$: the tag of the serial port the device is attached to

device_tag\$: the tag of the device

If you have requested exclusive access to any serial ports using the **GRAB** command, you cannot disable devices on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can disable devices on any port you like.

Disabling a device using the **DISABLEDRV** command is equivalent to selecting "Devices..." from the "Settings" menu and disabling it from within the Enable/Disable Devices dialog.

In device driver programs, port_tag\$ and device_tag\$ must be empty strings (" "). Only the device the program belongs to can be accessed by device driver programs.

If port_tag\$ is not the tag of a serial port, or device_tag\$ is not the tag of a device on that port, **DISABLEDRV** generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `DISABLEDRV` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `ENABLEDRV` Command

Enables a device on a serial port.

Syntax:

ENABLEDRV port_tag\$,device_tag\$

Arguments:

port_tag\$: the tag of the serial port the device is attached to

device_tag\$: the tag of the device

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot enable devices on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can enable devices on any port you like.

Enabling a device using the `DISABLEDRV` command is equivalent to selecting "Devices..." from the "Settings" menu and enabling it from within the Enable/Disable Devices dialog.

In device driver programs, port_tag\$ and device_tag\$ must be empty strings (" "). Only the device the program belongs to can be accessed by device driver programs.

If port_tag\$ is not the tag of a serial port, or device_tag\$ is not the tag of a device on that port, `ENABLEDRV` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `ENABLEDRV` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `DISABLECMD` Command

Disables a device command.

Syntax:

DISABLECMD port_tag\$,device_tag\$,command_tag\$, *command parameters*

Arguments:

port_tag\$: the tag of the serial port the command's device is attached to

device_tag\$: the tag of the command's device

command_tag\$: the tag of the command

The tag of the command is followed by a list of parameters, separated by commas. You can specify an expression for each of the command's parameters, in the order in which they are defined in the device driver. You should only specify values for parameters that

are used to distinguish between commands (see *Overview Of Commands* in the *Developing Device Drivers* manual).

Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

You can leave out one or more parameters from the end. If you leave out parameters, the command will be disabled for all possible values of that parameter.

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot disable commands on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can disable commands on any port you like.

Disabling a device command of a disabled device has no immediate effect. Once the device is re-enabled, however, commands disabled using the `DISABLECMD` will remain disabled, whereas enabled commands will become enabled.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only commands of the device the program belongs to can be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device driver on that port, or `command_tag$` is not the tag of a command in the device's driver, `DISABLECMD` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `DISABLECMD` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `ENABLECMD` Command

Enables a device command.

Syntax:

ENABLECMD `port_tag$,device_tag$,command_tag$, command parameters`

Arguments:

`port_tag$`: the tag of the serial port the command's device is attached to

`device_tag$`: the tag of the command's device

`command_tag$`: the tag of the command

The tag of the command is followed by a list of parameters, separated by commas. You can specify an expression for each of the command's parameters, in the order in which they are defined in the device driver. You should only specify values for parameters that are used to distinguish between commands (see *Overview Of Commands* in the *Developing Device Drivers* manual).

Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

You can leave out one or more parameters from the end. If you leave out parameters, the command will be enabled for all possible values of that parameter.

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot enable commands on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can enable commands on any port you like.

Enabling a device command of a disabled device has no immediate effect. Once the device is re-enabled, however, enabled commands will become enabled with it, whereas commands disabled using the `DISABLECMD` will remain disabled.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only commands of the device the program belongs to can be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `command_tag$` is not the tag of a command in the device's driver, `ENABLECMD` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `ENABLECMD` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SUSPEND` Command

Suspends polling on a serial port.

Syntax:

SUSPEND tag\$

Arguments:

tag\$: the tag of the serial port

You can suspend polling on any port at any time. Even if you have grabbed serial ports, you can still suspend the ports you have not grabbed.

If polling is already suspended on the port, this command has no effect.

In device driver programs, `tag$` must be an empty string (" "). Only the serial port of the device the program belongs to can be accessed by device driver programs.

If `tag$` is not the tag of a serial port, `SUSPEND` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `RESUME` Command

Resumes suspended polling on a serial port.

Syntax:

RESUME tag\$

Arguments:

tag\$: the tag of the serial port

You can resume polling on any port at any time. Even if you have grabbed serial ports, you can still resume the ports you have not grabbed.

If polling is not suspended on the port, this command has no effect.

In device driver programs, `tag$` must be an empty string (" "). Only the serial port of the device the program belongs to can be accessed by device driver programs.

If `tag$` is not the tag of a serial port, `RESUME` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Serial Device Object Commands

■ The `SETDRVOBJVAL` Command

Sets one value of a serial data object.

Syntax:

SETDRVOBJVAL `port_tag$,device_tag$,object_tag$, object parameters,value`

or

SETDRVOBJVAL `port_tag$,device_tag$,object_tag$, object parameters,value$`

or

SETDRVOBJVAL `port_tag$,device_tag$,object_tag$, object parameters,value%`

or

SETDRVOBJVAL `port_tag$,device_tag$,object_tag$,\
object parameters,value,index`

Arguments:

`port_tag$:` the tag of the serial port the data object's device is attached to

`device_tag$:` the tag of the data object's device

`object_tag$:` the tag of the data object

`value, value$, value%:` the value to set

`index:` the 1-based index of the value

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`SETDRVOBJVAL` sets the value of a serial data object. For analog data objects, it sets a single value of an analog data object, using "normal" (equal) as the greater / less status. To set an analog value with a different greater / less status, use `SETDRVOBJVALGL`.

For bistate data objects, specify a Boolean expression (`value%`) for the value. If `value%` is true, the data object will go into its ON. If `value%` is false, it will go into its OFF state. For digi-

tal data objects, either specify a numerical expression (`value`) for the value directly, or a string expression (`value$`) for the value's name. For analog data objects, specify a numerical expression (`value`), for string data objects a string expression (`value$`) for the value.

If the data object is not masked, all other data objects and registers that depend on it will be updated, if necessary.

For analog objects you can specify the index of the value to set. A data object with a size of one value has only a value with index 1. A data object with a size of 4 values has values with indices 1, 2, 3, and 4.

If an analog data object has a size of one value, you do not have to specify an index, as it is always 1. If the data object has a size of more than one value, and you do not specify an index, then all existing values will be shifted to the next lower index, and the value with the highest index will be set to `value`. This allows you to create a history of values by calling `SETDRVOBJVAL` repeatedly at regular intervals.

If the data object was in the error state, *all* values will be set to `value`.

To set all the values of an analog data object with a size of more than one value at once, use the `SETDRVOBJVALS` command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (""). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a data object in the device's driver, `SETDRVOBJVAL` generates an error.

If you specified the wrong type of expression for the type of data object, or if `object_tag$` is the not tag of an analog data object and you specified an index, `SETDRVOBJVAL` generates an error.

If `index` is smaller than 1, larger than the size of the data object, or if it contains fractions, `SETDRVOBJVAL` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **SETDRVOBJVALGL** Command

Sets one value and its greater / less status of an analog serial data object.

Syntax:

SETDRVOBJVALGL

```
port_tag$, device_tag$, object_tag$, value, greater_less_status
```

or

SETDRVOBJVALGL

```
port_tag$, device_tag$, object_tag$, value, greater_less_status, index
```

Arguments:

port_tag\$:	the tag of the serial port the data object's device is attached to
device_tag\$:	the tag of the data object's device
object_tag\$:	the tag of the data object
value:	the value to set
greater_less_status:	the greater / less status of the value
index:	the 1-based index of the value

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

SETDRVOBJVALGL sets a single value of an analog serial data object. The greater / less status of the value will be set to "less than" if `greater_less_status` is less than 0, and to "greater than" if it is greater than 0. If `greater_less_status` is equal to zero, the value will be a normal value (equal). To set the value to a normal value, you can also use the **SETDRVOBJVAL** command.

`index` is the index of the value. A data object with a size of one value has only a value with index 1. A data object with a size of 4 values has values with indices 1, 2, 3, and 4.

If the data object has a size of one value, you do not have to specify an index, as it is always 1. If the data object has a size of more than one value, and you do not specify an index, then all existing values will be shifted to the next lower index, and the value with the highest index will be set to `value`. This allows you to create a history of values by calling **SETDRVOBJVALGL** repeatedly at regular intervals.

If the data object was in the error state, *all* values will be set to `value` and `greater_less_status`.

If the data object is not masked, all other data objects and registers that depend on it will be updated, if necessary.

To set all the values and greater / less statuses of an analog data object with a size of more than one value at once, use the `SETDRV OBJVALSGL` command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of an analog data object in the device's driver, `SETDRV OBJVALGL` generates an error.

If `index` is smaller than 1, larger than the size of the data object, or if it contains fractions, `SETDRV OBJVALGL` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETDRV OBJVALS` Command

Sets all the values of an analog serial data object at once.

Syntax:

```
SETDRV OBJVALS port_tag$,device_tag$,object_tag$, \
                object_parameters,value_array
```

Arguments:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object
`value_array`: an array containing the values to set

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

`SETDRVOBJVALS` sets all the values of an analog serial data object at the same time, using "normal" (equal) as the greater / less status. To set values with different greater / less statuses, use `SETDRVOBJVALSGL`.

`value_array` must be a 1-dimensional array variable that contains the new values. `value_array[1]` contains the value with index 1, `value_array[2]` contains the value with index 2, etc.. The index of the first value is 1.

Do not specify an index for `value_array`. Only use the array name, without subscript.

If the data object is not masked, all other data objects and data objects that depend on it will be updated, if necessary.

To set a single value of an analog data object, use the `SETDRVOBJVAL` command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of an analog data object in the device's driver, `SETDRVOBJVALS` generates an error.

If `value_array` is not a 1-dimensional array, `SETDRVOBJVALS` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETDRVOBJVALSGL` Command

Sets all the values and greater / less status of an analog serial data object at once.

Syntax:

```
SETDRVOBJVALSGL port_tag$,device_tag$,object_tag$, \
    object_parameters,value_array,greater_less_array
```

Arguments:

<code>port_tag\$:</code>	the tag of the serial port the data object's device is attached to
<code>device_tag\$:</code>	the tag of the data object's device
<code>object_tag\$:</code>	the tag of the data object
<code>value_array:</code>	an array containing the value to set
<code>greater_less_array:</code>	an array containing the greater / less status of the values

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for

bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

SETDRVOBJVALS sets all the values of an analog serial data object at the same time.

`value_array` and `greater_less_array` must be a 1-dimensional array variable that contains the new values and their greater / less statuses. `value_array[1]` and `greater_less_array[1]` contain the information for the value with index 1, `value_array[2]` and `greater_less_array[2]` contain the information for the value with index 2, etc.. The index of the first value is 1.

`value_array` contains the actual values. `greater_less_array` contains the greater less flags. The greater / less status of a value will be set to "less than" if the corresponding element of `greater_less_array` is less than 0, and to "greater than" if it is greater than 0. If an element is equal to zero, the corresponding value will be a normal value (equal). To set all the values to normal values, you use the SETDRVOBJVALS command.

Do not specify an index for `value_array` or `greater_less_array`. Only use the array names, without subscript.

If the data object is not masked, all other data objects and data objects that depend on it will be updated, if necessary.

To set a single value and greater / less status of an analog data object, use the SETDRVOBJVALGL command.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of an analog data object in the device's driver, SETDRVOBJVALS generates an error.

If `value_array` or `greater_less_array` is not a 1-dimensional array, SETDRVOBJVALSGL generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The MASKDRVOBJ Command

Masks a serial data object.

Syntax:

MASKDRVOBJ `port_tag$,device_tag$,object_tag$, object parameters`

Arguments:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog

parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

You can leave out one or more parameters from the end. If you leave out parameters, the data object will be unmasked for all possible values of that parameter.

Masking a data object using the `MASKDRVOBJ` command applies a special mask to the data object. `MASKDRVOBJ` is independent of automatic masking.

Use `UNMASKDRVOBJ` to remove the special mask.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a data object in the device's driver, `MASKDRVOBJ` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `UNMASKDRVOBJ` Command

Unmasks a serial data object.

Syntax:

`UNMASKDRVOBJ` `port_tag$,device_tag$,object_tag$, object parameters`

Arguments:

`port_tag$`: the tag of the serial port the data object's device is attached to
`device_tag$`: the tag of the data object's device
`object_tag$`: the tag of the data object

The tag of the data object is followed by a list of parameters, separated by commas. You must specify an expression for each of the object's parameters, in the order in which they are defined in the device driver. Use a numerical expression for digital and analog parameters, a string expression for string parameters, and a Boolean expression for bistate parameters. You can also use a string expression to specify the name of the value for digital parameters.

You can leave out one or more parameters from the end. If you leave out parameters, the data object will be unmasked for all possible values of that parameter.

Unmasking a data object using the `UNMASKDRVOBJ` command removes the special mask applied using the `MASKDRVOBJ` command. `UNMASKDRVOBJ` is independent of manual masking and of automatic masking. To unmask a data object as if unmasked manually by the operator, use the `UNMASK` command instead.

In device driver programs, `port_tag$` and `device_tag$` must be empty strings (" "). Only data objects of the device the program belongs to can be accessed by device driver programs.

Internal data objects can only be accessed by device driver programs.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `object_tag$` is not the tag of a data object in the device's driver, `UNMASKDRVOBJ` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Logging Commands

■ The `LOG` Command

Writes text to the log file and log window.

Syntax:

LOG *list of expressions*

`LOG` is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are written to the log directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the log.

The log entry is preceded by the current time and date.

`LOG` writes the message to both the log file and the log window. You can log to the log file only using the `FILELOG` command.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `LOGR` Command

Writes red text to the log file and log window.

Syntax:

LOGR *list of expressions*

This command behaves just like the `LOG` command, but the log message will appear in red in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `LOGG` Command

Writes green text to the log file and log window.

Syntax:**LOGG** *list of expressions*

This command behaves just like the LOG command, but the log message will appear in green in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The LOGB Command

Writes blue text to the log file and log window.

Syntax:**LOGB** *list of expressions*

This command behaves just like the LOG command, but the log message will appear in blue in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The LOGC Command

Writes cyan (sky blue) text to the log file and log window.

Syntax:**LOGC** *list of expressions*

This command behaves just like the LOG command, but the log message will appear in cyan in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The LOGM Command

Writes magenta text to the log file and log window.

Syntax:**LOGM** *list of expressions*

This command behaves just like the LOG command, but the log message will appear in magenta in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The LOGY Command

Writes yellow (actually, orange) text to the log file and log window.

Syntax:**LOGY** *list of expressions*

This command behaves just like the LOG command, but the log message will appear in yellow in the log window and the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The FILELOG Command

Writes text to the log file, but not to the log window.

Syntax:

FILELOG *list of expressions*

FILELOG is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are written to the log file directly. Numbers are written in decimal format, using exponential notation whenever necessary. Boolean values are written as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a tab in the log file.

The log file entry is preceded by the current time and date.

FILELOG writes the message to the log file only, and not to the log window. To log the message to the log window as well, use the LOG command.

Unlike the PROMPT command, you must not end the list of expressions with a semicolon or comma.

This command is not available within programs for sources, checksums, and SABus response data.

■ The FILELOGR Command

Writes red text to the log file, but not to the log window.

Syntax:

FILELOGR *list of expressions*

This command behaves just like the FILELOG command, but the log message will appear in red in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The FILELOGG Command

Writes green text to the log file, but not to the log window.

Syntax:

FILELOGG *list of expressions*

This command behaves just like the FILELOG command, but the log message will appear in green in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **FILELOGB** Command

Writes blue text to the log file, but not to the log window.

Syntax:

FILELOGB *list of expressions*

This command behaves just like the **FILELOG** command, but the log message will appear in blue in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **FILELOGC** Command

Writes cyan (sky blue) text to the log file, but not to the log window.

Syntax:

FILELOGC *list of expressions*

This command behaves just like the **FILELOG** command, but the log message will appear in cyan in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **FILELOGM** Command

Writes magenta text to the log file, but not to the log window.

Syntax:

FILELOGM *list of expressions*

This command behaves just like the **FILELOG** command, but the log message will appear in magenta in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

■ The **FILELOGY** Command

Writes yellow (actually, orange) text to the log file, but not to the log window.

Syntax:

FILELOGY *list of expressions*

This command behaves just like the **FILELOG** command, but the log message will appear in yellow in the log file browser.

This command is not available within programs for sources, checksums, and SABus response data.

Data Encoding/Decoding Commands

■ The **PARSEDEC** Command

Decodes a value contained within a string using a decoder and a position variable.

Syntax:

PARSEDEC string\$,decoder,result_variable,position_variable

or

PARSEDEC string\$,decoder,result_variable\$,position_variable

or

PARSEDEC string\$,decoder,result_variable%,position_variable

Arguments:

string\$:	the data string that contains the value
decoder:	the number of the decoder to use
result_variable:	a variable to receive the decoded number
result_variable\$:	a variable to receive the decoded string
result_variable%:	a variable to receive the decoded Boolean value
position_variable:	the variable that contains the current position within the string

PARSEDEC decodes a value from `string$`, starting with the character at the position contained in `position_variable`. After the value has been decoded, `position_variable` is set to the position of the first character after the value. This allows you to then call **PARSEDEC** again to decode another value using the same position variable.

The first character in a string is position number 1. If you set `position_variable` to 0 before calling **PARSEDEC**, parsing will start at the first character in the string (same as `position_variable = 1`).

If the value was not found at the specified position, **PARSEDEC** sets `position_variable` to 0.

To parse a string value using a decoder you create on the fly instead of a pre-defined one, use the **PARSEREGEX** command. To skip over an encoded value without storing it in a variable, use the **SKIPDEC** command.

To decode a single value from a string, use the **DECODE**, **DECODE\$**, or **DECODE%** function.

If `decoder` is not the number of a decoder for the right type of value, or if the value of `position_variable` is less than 0 or contains fractions, **PARSEDEC** generates an error.

■ The **PARSEREGEX** Command

Decodes a string value contained within a string using three regular expressions and a position variable.

Syntax:

PARSEREGEX

string\$,prefix\$,pattern\$,suffix\$,result_variable\$,position_variable

Arguments:

<code>string\$:</code>	the data string that contains the value
<code>prefix\$:</code>	the prefix pattern, or "" for none
<code>pattern\$:</code>	the data pattern
<code>suffix\$:</code>	the suffix pattern, or "" for none
<code>result_variable\$:</code>	a variable to receive the decoded string
<code>position_variable:</code>	the variable that contains the current position within the string

The prefix pattern, if any, describes any data that appears in `string$` before the encoded string. The data pattern describes the encoded string itself, and the suffix pattern, if any, describes data that must appear after the encoded string (e.g. a terminating character).

See *Appendix A: Regular Expressions* in the *Developer's Manual* for details on regular expressions.

`PARSEREGEX` decodes a string value from `string$`, starting with the character at the position contained in `position_variable`. After the value has been decoded, `position_variable` is set to the position of the first character after the value. This allows you to then call `PARSEREGEX` again to decode another value using the same position variable.

The first character in a string is position number 1. If you set `position_variable` to 0 before calling `PARSEREGEX`, parsing will start at the first character in the string (same as `position_variable = 1`).

If the value was not found at the specified position, `PARSEREGEX` sets `position_variable` to 0.

To parse a value using a pre-defined decoder, use the `PARSEDEC` command. To skip over data without storing it in a variable, use the `SKIPREGEX` command.

To decode a single value from a string, use the `DECODEREGEX$` function.

If `pattern$` is not a valid regular expression, or if `prefix$` or `suffix$` is not an empty string or a valid regular expression, `PARSEREGEX` generates an error.

If the value of `position_variable` is less than 0, or if it contains fractions, `PARSEREGEX` generates an error.

■ The `SKIPDEC` Command

Skips over a value contained within a string using a decoder and a position variable.

Syntax:

`SKIPDEC` `string$,decoder,position_variable`

Arguments:

<code>string\$:</code>	the data string that contains the value
<code>decoder:</code>	the number of the decoder to use
<code>position_variable:</code>	the variable that contains the current position within the string

`SKIPDEC` skips over a value encoded in `string$` starting with the character at the position contained in `position_variable`. It sets `position_variable` to the position of the

first character after the value. This allows you to then call `PARSEDEC` to decode the next value using the same position variable.

The first character in a string is position number 1. If you set `position_variable` to 0 before calling `SKIPDEC`, it will look for the value at the first character in the string (same as `position_variable = 1`).

If the value was not found at the specified position, `SKIPDEC` sets `position_variable` to 0.

To skip arbitrary data using a regular expression instead of a pre-defined decoder, use the `SKIPREGEX` command. To decoded a value and store it in a variable, use the `PARSEDEC` command.

If `decoder` is not the number of a decoder, or if the value of `position_variable` is less than 0 or contains fractions, `SKIPDEC` generates an error.

■ The `SKIPREGEX` Command

Skips arbitrary data in a string using a regular expression and a position variable.

Syntax:

SKIPREGEX `string$,regex$,position_variable`

Arguments:

`string$`: the data string that contains the value
`regex$`: the regular expression
`position_variable`: the variable that contains the current position within the string

See *Appendix A: Regular Expressions* in the *Developer's Manual* for details on regular expressions.

`SKIPREGEX` skips over all data in `string$` that matches `regex$`, starting with the character at the position contained in `position_variable`. It sets `position_variable` to the position of the first character after the value. This allows you to then call `PARSEDEC` or `PARSEREGEX` to decode a value that appears after the data using the same position variable.

The first character in a string is position number 1. If you set `position_variable` to 0 before calling `SKIPREGEX`, it will look for the data at the first character in the string (same as `position_variable = 1`).

If matching data was not found at the specified position, `SKIPREGEX` sets `position_variable` to 0.

To skip an encoded value using a decoder, use the `SKIPDEC` command. To decoded a value and store it in a variable, use the `PARSEDEC` or `PARSEREGEX` command.

If `regex$` is not a valid regular expression, or if the value of `position_variable` is less than 0 or contains fractions, `SKIPREGEX` generates an error.

■ The `APPENDSTR` Command

Appends one or more strings to the value of a string variable.

Syntax:

APPENDSTR *string_variable\$,string1\$,string2\$, etc.*

Arguments:

string_variable\$: the string variable to append data to
string1\$, string2\$, etc.: the strings to append

■ The **APPENDCSTR** Command

Appends one or more strings to the value of a string variable. The strings can contain non-printable characters encoded in special backslash sequences similar to those C compilers use.

Syntax:

APPENDCSTR *string_variable\$,C_string1\$,C_string2\$, etc.*

Arguments:

string_variable\$: the string variable to append data to
C_string1\$, C_string2\$, etc.: the C style strings to append

APPENDCSTR enables you to easily specify strings containing non-printable characters (ASCII 00-1F and 7F-FF). To specify a non-printable character, use any of the following sequences of characters:

Sequence	Character	Code (hexadecimal)
\0	null character	\$00
\b	backspace	\$08
\t	tab	\$09
\n	linefeed	\$0A
\v	vertical tab	\$0B
\f	form feed	\$0C
\r	carriage return	\$0D

to specify any other non-printable character, use `\x` followed by two hexadecimal digits specifying the character code. Here are some examples:

Sequence	Character	Code (hexadecimal)
\x02	start transmission	\$02
\x03	end of transmission	\$03
\xFF	delete	\$FF
\xB7	\$B7	
\x69	capital letter "E"	\$69

To specify a backslash, use two backslashes in a row:

Sequence	Character	Code (hexadecimal)
\\	backslash	\$92

Printable characters, with the exception of the backslash and double quotes, can just be entered plainly. If you feel so inclined, however, you can use a backslash followed by that character.

Here are some examples:

Sequence	Character	Code (hexadecimal)
\a	letter "a"	\$97
\6	digit six	\$56
\/	slash	\$47
\R	capital letter "R"	\$82

To convert a single C style string, use the `CCNV$` function.

Note: `APPENDCSTR` would theoretically convert `\"` to the double quote character if it encountered it in a string. However, since the SCL interpreter will not allow double quotes in string literals, you *cannot* use the `\"` sequence to specify double quotes. To generate the string:

```
Hi! My name is Fred "Barbarossa" Staufer!
```

You cannot use the following command:

```
APPENDCSTR string_variable$, \
  "Hi! My name is Frederick \"Barbarossa\" Staufer!" ← error!
```

You can use the `\x` character sequence and specify the ASCII code for the double quotes character instead:

```
APPENDCSTR string_variable$, \
  "Hi! My name is Frederick \x34Barbarossa\x34 Staufer!"
```

You can also use the `QUT$` constant with `APPENDSTR` instead of `APPENDCSTR`:

```
APPENDSTR string_variable$, \
  "Hi! My name is Frederick ", QUT$, "Barbarossa", QUT$, \
  " Staufer!"
```

If `C_string$` ends in a backslash, or if it contains a `\x` that is not followed by two hexadecimal digits, `APPENDCSTR` generates an error.

■ The **APPENDHEX** Command

Appends one or more strings consisting of arbitrary character codes written out in hexadecimal to the value of a string variable.

Syntax:

APPENDHEX *string_variable\$,hex_values1\$,hex_values2\$,hex_values3\$, etc.*

Arguments:

string_variable\$: the string variable to append data to
hex_values1\$, hex_values2\$, etc.: the strings with hex values are written out to append

The hex value strings have to be strings consisting of two digit hexadecimal values separated by single spaces. **APPENDHEX** would append the string:

```
"4F 7A 6F 6E 65 21"
```

as the following string:

```
"Ozone!"
```

To convert a single hex value string, use the **HCNV\$** function.

If any of the hex value strings do not conform to the format described above, or if there are any characters before the first or after the last hex value (including spaces), **APPENDHEX** generates an error.

■ The **APPENDENC** Command

Appends one or more values to the value of a string variable using a data encoder.

Syntax:

APPENDENC *string_variable\$,encoder,number1,number2, etc.*

or

APPENDENC *string_variable\$,encoder,string1\$,string2\$, etc.*

or

APPENDENC *string_variable\$,encoder,Boolean1%,Boolean2%, etc.*

Arguments:

string_variable\$: the string variable to append data to
encoder: the number of the encoder to use
number1\$, number2\$, etc.: the numbers to encode
string1\$, string2\$, etc.: the strings to encode
Boolean1%, Boolean2%, etc.: the Boolean values to encode

APPENDENC encodes the values using the specified encoding and appends them to the value of the string variable. No separators are placed between the different values.

To encode a single value to a string, use the **ENCODE\$** function.

If `encoder` is not the number of an encoder for the right type of value, `APPENDENC` generates an error.

Miscellaneous Commands

■ The **SETPVAR** Command

Sets the value of a variable of the parent program.

Syntax:

SETPVAR num_var_name\$, value

or

SETPVAR str_var_name\$, value\$

or

SETPVAR bool_var_name\$, value%

Arguments:

num_var_name\$: the name of a numerical variable of the parent program

str_var_name\$: the name of a string variable of the parent program

bool_var_name\$: the name of a Boolean variable of the parent program

value, value\$, value%: the value to set the variable to

The parent program is the program that called this program using the `CALL`, `DRVCALL`, `CALLRMT` or command.

`SETPVAR` cannot be used to access array elements.

If the name specified is not a valid name for a variable, if it is the name of a reserved SCL keyword, or if it is the name of an array variable, `SETPVAR` generates an error.

If the type of the value does not correspond to the type of the variable specified, `SETPVAR` generates an error.

This command is only available within child programs.

■ The **DELAY** Command

Waits for a specified amount of time.

Syntax:

DELAY seconds

or

DELAY seconds,result_variable%

Arguments:

seconds: the number of seconds to wait

result_variable%: a variable to receive the abort status

`DELAY` waits the specified number of seconds.

If you specify a result variable, then the wait will end immediately if the station is closed. `result_variable` will be true if the wait was successful, and false if the wait was aborted. You should specify a result variable for delays longer than about 2 seconds to allow the database to be closed while the program is running.

If `seconds` is not between 0 and 4,294,967.295, `DELAY` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `CALL` Command

Calls another program as a child program.

Syntax:

CALL tag\$

or

CALL tag\$, *list of arguments and values*

Arguments:

tag\$: the tag of the program to call

If you use `CALL` within a normal program (defined outside a device driver), it will call a normal program. If you use `CALL` from within a device driver program, it will call a device driver program from the same port and device.

`CALL` will wait until the child program has finished executing. If an error occurs in the child program, `CALL` will end the parent program.

All commands and functions that are available within the parent program are also available within the child. A child program of an RTS control, for example, can use the `RTSPRM$` function, and the `RTSSEND` and `RTSERROR` commands. Commands that are not available within the parent, are also not available from within the child. You cannot use the `PRINT` command, for example, in a program that was called by a program used in a processor source.

Child programs of programs for processor and summary sources have the default limit of 500 instructions that their parents have.

Child programs can only use the `GRAB` command if the parent does not have exclusive access to any serial ports. If the parent does have exclusive access to any ports, the child program can access only those ports. Child programs cannot release ports the parent has grabbed, and any ports the child has grabbed will be released when it ends.

`CALL` can include an argument list, to be used as program arguments for the child program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If tag\$ is not the tag of an SCL program, `CALL` generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, `CALL` generates an error.

■ The **DRVCALL** Command

Calls a device driver program as a child program.

Syntax:

DRVCALL port_tag\$,device_tag\$,program_tag\$

or

DRVCALL port_tag\$,device_tag\$,program_tag\$, *list of arguments and values*

Arguments:

port_tag\$: the tag of the serial port the program's device is attached to

device_tag\$: the tag of the device whose driver contains the program

program_tag\$: the tag of the program to call

DRVCALL will call a device driver program. If you use **DRVCALL** from within a device driver program, port_tag\$ and device_tag\$ must be empty strings (" "). Only programs of the device the parent program belongs to can be called by device driver programs. You should use **CALL** instead of **DRVCALL** within device driver programs.

DRVCALL will wait until the child program has finished executing. If an error occurs in the child program, **DRVCALL** will end the parent program.

All commands and functions that are available within the parent program are also available within the child. A child program of an RTS control, for example, can use the **RTSPRM\$** function, and the **RTSEND** and **RTSError** commands. Commands that are not available within the parent, are also not available from within the child. You cannot use the **PRINT** command, for example, in a program that was called by a program used in a processor source.

Child programs of programs for processor and summary sources have the default limit of 500 instructions that their parents have.

Child programs can only use the **GRAB** command if the parent does not have exclusive access to any serial ports. If the parent does have exclusive access to any ports, the child program can access only those ports. Child programs cannot release ports the parent has grabbed, and any ports the child has grabbed will be released when it ends.

DRVCALL can include an argument list, to be used as program arguments for the child program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If port_tag\$ is not the tag of a serial port, device_tag\$ is not the tag of a device on that port, or program_tag\$ is not the tag of an SCL program in the device's driver, **DRVCALL** generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, **DRVCALL** generates an error.

■ The **CALLRMT** Command

Calls a program on a remote computer as a child program.

Syntax:

CALLRMT ip_address\$,tag\$

or

CALLRMT ip_address\$,tag\$, *list of arguments and values*

or

CALLRMT ip_address\$,tag\$,result_variable%

or

CALLRMT ip_address\$,tag\$, *list of arguments and values*,result_variable%

Arguments:

ip_address\$: the IP address of the server

tag\$: the tag of the program to call

result_variable%: a variable to receive the success/failure status

ip_address\$ is the remote computer's IP address in dotted form, e.g. "192.168.1.100". The first component of an IP address cannot be 0, 127, or greater than 223, except for the loopback address 127.0.0.1.

CALLRMT attempts connect to another computer running U.P.M.A.C.S. and call a program on that computer. If you do not specify a result variable, CALLRMT will end the program if the connection cannot be established. If you do specify a result variable, it will be set to true if the connection could be established, and false if it could not.

CALLRMT will wait until the child program has finished executing. If an error occurs in the child program, CALLRMT will end the parent program. The programs executed using CALLRMT run on the remote computer, and have access to the serial ports, registers, etc. of the remote computer's local station, not that of the parent program's computer.

CALLRMT can only run programs on a remote computer if the following apply:

- the remote computer is accessible via TCP/IP
- U.P.M.A.C.S. is running on the remote computer
- a local station file has been loaded on the remote computer
- remote connections are enabled in the remote computer's network security settings
- insecure remote control is enabled in the remote computer's network security settings
- remote connections have not been disabled using the STOPNET command on the remote computer

Contrary to the CALL command, a program started using the CALLRMT command does *not* inherit access to any serial ports from the program that invoked it. Programs run from RTS controls *cannot* use the RTSPRM\$ function, or the RTSEND and RTSERROR commands. However, commands that are not available within the parent program are also not available within the remote child program. Programs run from SABus command programs and RTS controls therefore cannot use user message or dialog commands.

Access to the `GRAB` command is not limited in the same way it is for programs executed using `CALL`. Any program executed using `CALLRMT` can grab serial ports, regardless of whether the parent has grabbed ports or not.

Programs executed using the `CALLRMT` command can access the parent program's user variables using the `SETPVAR` command and the `PVAR`, `PVAR$`, and `PVAR%` functions.

Any dialogs or user messages that the remote program pops up will be shown on the same computer as those of the parent program (usually the local computer of the parent program).

`CALLRMT` can include an argument list, to be used as program arguments for the child program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If `ip_address$` is not a valid IP address, or if `tag$` is not the tag of an SCL program on the remote computer, `CALLRMT` generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, `CALLRMT` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `RUN` Command

Runs another program as an independent program.

Syntax:

RUN tag\$

or

RUN tag\$,delay_seconds

or

RUN tag\$, list of arguments and values

or

RUN tag\$, list of arguments and values,delay_seconds

Arguments:

tag\$: the tag of the program to run

delay_seconds: the amount of time to wait before running the program (0 to 4,294,967.295s)

If you use `RUN` within a normal program (defined outside a device driver), it will run a normal program. If you use `RUN` from within a device driver program, it will run a device driver program from the same port and device.

`RUN` will run the specified program as an independent program. It does not wait until the programs has finished, but returns immediately.

Contrary to the `CALL` command, a program started using the `RUN` command does *not* inherit access to any serial ports from the program that invoked it. Programs run from RTS controls *cannot* use the `RTSPRM$` function, or the `RTSEND` and `RTSERROR` commands.

`RUN` can include an argument list, to be used as program arguments for the new program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If `tag$` is not the tag of an SCL program, or if `delay_seconds` is not between 0 and 4,294,967.295, `RUN` generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, `RUN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `DRVRUN` Command

Runs a device driver program as an independent program.

Syntax:

DRVRUN port_tag\$,device_tag\$,program_tag\$

or

DRVRUN port_tag\$,device_tag\$,program_tag\$,delay_seconds

or

DRVRUN port_tag\$,device_tag\$,program_tag\$, *list of arguments and values*

or

DRVRUN port_tag\$,device_tag\$,program_tag\$, *list of arguments and values*,delay_seconds

Arguments:

port_tag\$: the tag of the serial port the program's device is attached to

device_tag\$: the tag of the device whose driver contains the program

program_tag\$: the tag of the program to run

delay_seconds: the amount of time to wait before running the program (0 to 4,294,967.295s)

`DRVRUN` will run a device driver program. If you use `DRVRUN` from within a device driver program, `port_tag$` and `device_tag$` must be empty strings (" "). Only programs of the device the parent program belongs to can be run by device driver programs. You should use `RUN` instead of `DRVRUN` within device driver programs.

`DRVRUN` will run the specified program as an independent program. It does not wait until the programs has finished, but returns immediately.

Contrary to the `DRVCALL` command, a program started using the `DRVRUN` command does *not* inherit access to any serial ports from the program that invoked it. Programs run

from RTS controls *cannot* use the `RTSPRM$` function, or the `RTSEND` and `RTSErrorR` commands.

`DRVRUN` can include an argument list, to be used as program arguments for the new program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on that port, or `program_tag$` is not the tag of an SCL program in the device's driver, `DRVDRVRUN` generates an error.

If `delay_seconds` is not between 0 and 4,294,967.295, `DRVRUN` generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, `DRVRUN` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `RUNRMT` Command

Runs a program on a remote computer as a child program.

Syntax:

RUNRMT `ip_address$,tag$`

or

RUNRMT `ip_address$,tag$, list of arguments and values`

or

RUNRMT `ip_address$,tag$,result_variable%`

or

RUNRMT `ip_address$,tag$, list of arguments and values,result_variable%`

Arguments:

`ip_address$:` the IP address of the server

`tag$:` the tag of the program to run

`result_variable%:` a variable to receive the success/failure status

`ip_address$` is the remote computer's IP address in dotted form, e.g. "192.168.1.100". The first component of an IP address cannot be 0, 127, or greater than 223, except for the loopback address 127.0.0.1.

`RUNRMT` attempts connect to another computer running U.P.M.A.C.S. and run a program on that computer. If you do not specify a result variable, `RUNRMT` will end the program if the connection cannot be established. If you do specify a result variable, it will be set to true if the connection could be established, and false if it could not.

`RUNRMT` will run the specified program as an independent program. It does not wait until the programs has finished, but returns immediately.

`RUNRMT` can only run programs on a remote computer if the following apply:

- the remote computer is accessible via TCP/IP
- U.P.M.A.C.S. is running on the remote computer
- a local station file has been loaded on the remote computer
- remote connections are enabled in the remote computer's network security settings
- insecure remote control is enabled in the remote computer's network security settings
- remote connections have not been disabled using the `STOPNET` command on the remote computer

Any dialogs or user messages that the remote program pops up will be shown on the same computer as those of the parent program (usually the local computer of the parent program).

`RUN` can include an argument list, to be used as program arguments for the new program. The argument list consists of pairs of expressions, specifying the argument's name and its value. The first expression in the pair has to be a string, and has to specify a valid SCL variable name, including the type suffix. The second expression has to specify a value of the correct type for the argument.

If `ip_address$` is not a valid IP address, or if `tag$` is not the tag of an SCL program on the remote computer, `RUNRMT` generates an error.

If an argument name is not a valid SCL variable name, if any argument name appears twice, or if the type any of the argument values does not match the type of the argument, `RUNRMT` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `LAUNCH` Command

Launches a Windows application.

Syntax:

LAUNCH `command_line$`

Arguments:

`command_line$`: the command line of the program

`LAUNCH` does not wait until the application has finished, but returns immediately.

If Windows reports that the application specified by `command_line$` does not exist or cannot be launched for other reasons, `LAUNCH` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `STOPNET` Command

Disables remote connections.

Syntax:**STOPNET**

STOPNET closes all remote connections from other computers, including connections to the local station, insecure remote control, and remote register source connection, and disables further connections.

If networking is already disabled, STOPNET has no effect.

This command is not available within programs for sources, checksums, and SABus response data.

■ The STARTNET Command

Enables remote connections.

Syntax:**STARTNET**

STARTNET enables remote connections if they have been enabled using STOPNET. If networking has not been disabled using STOPNET, STARTNET has no effect. STARTNET does not alter the network security settings, and does not enable networking if it is disabled in the network security settings.

This command is not available within programs for sources, checksums, and SABus response data.

Special Purpose Commands

■ The SABUSREPLY Command

Sends a reply to an uplink port that executed an SABus command.

Syntax:**SABUSREPLY**

or

SABUSREPLY *list of expressions*

Programs of SABus commands use the SABUSREPLY command to send the response to the uplink port that invoked the command. If necessary, you can specify data to be included in the reply.

SABUSREPLY may be followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are included in the response as is. Numbers are written out in decimal format, using exponential notation whenever necessary. Boolean values are included as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a space character.

It is not necessary to specify the packet header and footer for the response packet in the command. SABUSREPLY automatically adds the ACK, address, opcode, ETX, and checksum to the data you specify.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

Once you used the `SABUSREPLY` or `SABUSERERROR` commands, the uplink port is released, and you may not use `SABUSREPLY` or `SABUSERERROR` again. If you already used `SABUSREPLY` or `SABUSERERROR` to respond to the command, `SABUSREPLY` generates an error.

If any expression in the list contains non-printable characters (ASCII \$00-\$1F or \$7F-\$FF), `SABUSREPLY` generates an error.

This command is only available within programs for SABus commands.

■ The `SABUSERERROR` Command

Sends a reply to an uplink port that executed an SABus command.

Syntax:

SABUSERERROR *list of expressions*

Programs of SABus commands use the `SABUSERERROR` command to send an error response to the uplink port that invoked the command.

`SABUSERERROR` is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are included in the response as is. Numbers are written out in decimal format, using exponential notation whenever necessary. Boolean values are included as "true" or "false."

Expressions separated by a semicolon are placed immediately next to each other, expressions separated by a comma in the list are separated by a space character.

It is not necessary to specify the packet header and footer for the response packet in the command. `SABUSERERROR` automatically adds the `NAK`, address, opcode, `ETX`, and checksum to the data you specify.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

Once you used the `SABUSERERROR` or `SABUSREPLY` commands, the uplink port is released, and you may not use `SABUSERERROR` or `SABUSREPLY` again. If you already used `SABUSERERROR` or `SABUSREPLY` to respond to the command, `SABUSERERROR` generates an error.

If any expression in the list contains non-printable characters (ASCII \$00-\$1F or \$7F-\$FF), `SABUSERERROR` generates an error.

This command is only available within programs for SABus commands.

■ The `RTSSEND` Command

Sends a message to the computer that invoked an RTS control.

Syntax:

RTSSEND *list of expressions*

`RTSSEND` is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are sent directly. Numbers are

sent in decimal format, using exponential notation whenever necessary. Boolean values are sent as "true" or "false."

Expressions separated by a semicolon are sent immediately next to each other, expressions separated by a comma are sent with a tab separating them.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

This command is only available within RTS controls.

■ The `RTSError` Command

Sends a user error message to the computer that invoked an RTS control.

Syntax:

RTSError *list of expressions*

`RTSError` is followed by a list of expressions of any type (numerical, string, or Boolean), separated by commas or semicolons. String expressions are sent directly. Numbers are sent in decimal format, using exponential notation whenever necessary. Boolean values are sent as "true" or "false."

Expressions separated by a semicolon are sent immediately next to each other, expressions separated by a comma are sent with a tab separating them.

Unlike the `PROMPT` command, you must not end the list of expressions with a semicolon or comma.

This command is only available within RTS controls.

Legacy Object Commands

■ The `SENDREPLY` Command

Sends a pre-defined reply to a serial device that uses a legacy device driver.

Syntax:

SENDREPLY `port_tag$,device_tag$,reply_tag$`

Arguments:

`port_tag$`: the tag of the serial port the reply's device is attached to
`device_tag$`: the tag of the device to send the reply to
`reply_tag$`: the tag of the reply

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot send replies to any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can send replies to any port you like.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on the port that uses a legacy device driver, or `reply_tag$` is not the tag of a reply in that driver, `SENDREPLY` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `SENDREPLY` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `DISABLEMSG` Command

Disables a legacy device driver message for a device.

Syntax:

DISABLEMSG port_tag\$,device_tag\$,message_tag\$

Arguments:

port_tag\$: the tag of the serial port the message's device is attached to

device_tag\$: the tag of the device of the message

message_tag\$: the tag of the message

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot disable messages on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can disable messages on any port you like.

Disabling a device message of a disabled device has no immediate effect. Once the device is re-enabled, however, messages disabled using the `DISABLEMSG` will remain disabled, whereas enabled messages will become enabled.

If port_tag\$ is not the tag of a serial port, device_tag\$ is not the tag of a device on the port that uses a legacy device driver, or message_tag\$ is not the tag of a message in that driver, `DISABLEMSG` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `DISABLEMSG` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `ENABLEMSG` Command

Enables a legacy device driver message for a device.

Syntax:

ENABLEMSG port_tag\$,device_tag\$,message_tag\$

Arguments:

port_tag\$: the tag of the serial port the message's device is attached to

device_tag\$: the tag of the device of the message

message_tag\$: the tag of the message

If you have requested exclusive access to any serial ports using the `GRAB` command, you cannot enable messages on any ports other than the ones you have exclusive access to. If you do not have exclusive access to any ports, you can enable messages on any port you like.

Enabling a device message of a disabled device has no immediate effect. Once the device is re-enabled, however, enabled messages will become enabled with it, whereas messages disabled using the `DISABLEMSG` will remain disabled.

If `port_tag$` is not the tag of a serial port, `device_tag$` is not the tag of a device on the port that uses a legacy device driver, or `message_tag$` is not the tag of a message in that driver, `ENABLEMSG` generates an error.

If you have exclusive access to any serial ports, but not to the specified serial port, `ENABLEMSG` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

■ The `SETPARAM` Command

Sets the value of a legacy parameter.

Syntax:

SETPARAM tag\$,data

or

SETPARAM tag\$,data\$

or

SETPARAM tag\$,data%

Arguments:

tag\$: the tag of the parameter

data, data\$, data%: the data to set the parameter to

`SETPARAM` can be used to set the value of a parameter.

If the data is a string, it is copied exactly into the parameter's value.

If the data is a number, it is written out in the parameter in a fashion suitable for the `PARAM` function and for parameter sources of analog and digital registers.

If the data is a Boolean, the parameter's value is set to "ON" if `data%` is true, and to "OFF" if it is false. This is suitable for the `PARAM$` function and for parameter sources of bistate registers.

If `tag$` is not the tag of a parameter, `PARAM$` generates an error.

This command is not available within programs for sources, checksums, and SABus response data.

Obsolete Commands

These commands are obsolete and should not be used:

- The `INPUT` Command (Use dialogs instead.)

APPENDICES

Appendix A: Alphabetical List of Keywords

A

ABS
 ADDLITEM
 ADDMITEM
 ANAGL
 ANAHIGH
 ANALOW
 ANAMAX
 ANAMIN
 ANAVAL
 AND
 APPENDCSTR
 APPENDENC
 APPENDHEX
 APPENDSTR
 ASCII
 ASK

B

BCDFMT\$
 BCDVAL
 BEFMT\$
 BEVAL
 BINFMT\$
 BINVAL
 BSTDLY
 BSTVAL%
 BUFFER\$
 BUFFER
 BUTTON
 BUTTON0

C

CALL
 CALLRMT
 CANCELBTN
 CCNV\$
 CHKBOX
 CHKSUM\$
 CHKSUM
 CHKSUMHILO
 CHKSUMLOHI
 CHR\$
 CLOSE
 CLRANAMAX
 CLRANAMIN
 CLRLITEMS
 CLRMITEMS
 CMDENABLED%
 CONFIRM
 CONNECT
 COS
 COUNTLITEMS
 COUNTMITEMS
 CRC16
 CRC32
 CRCCCITT

D

DAY
 DECODE\$
 DECODE%
 DECODE
 DECODEREGEX\$
 DELAY
 DELLITEM
 DELMITEM
 DIALOG
 DIGVAL\$
 DIGVAL
 DISABLECMD
 DISABLEDRV
 DISABLEMSG
 DLGERROR
 DLGLINE
 DLGTEXT
 DLGTITLE
 DO
 DRVCALL
 DRVDATA\$
 DRVENABLED%
 DRVERROR\$
 DRVERROR%
 DRVERROR
 DRVNDATA\$
 DRVNERROR\$
 DRVNERROR%
 DRVOBJERR%
 DRVOBJGL
 DRVOBJHIGH
 DRVOBJLOW
 DRVOBJMASK%
 DRVOBJVAL\$
 DRVOBJVAL%
 DRVOBJVAL
 DRVPRM\$
 DRVPRM%
 DRVPRM
 DRVREADY%
 DRVRUN
 DRVSUCCESS%
 DRVTIMEOUT%

E

ELSE
 ELSEIF
 ENABLECMD
 ENABLEDRV
 ENABLEMSG
 ENCODE\$
 END
 ENDDO
 ENDIF
 ERRMSG
 EXP

F

FALSE%
 FILELOG
 FILELOGB
 FILELOGC
 FILELOGG
 FILELOGM
 FILELOGR
 FILELOGY
 FLEN
 FMT\$
 FOR
 FPOS

G

GMT
 GOSUB
 GOTO
 GRAB

H

HCNV\$
 HEXFMT\$
 HEXFMT2\$
 HEXVAL
 HIDE
 HILOFMT\$
 HILOVAL
 HRS

I

IF
 INFO
 INPUT#
 INPUT
 INTEDIT
 INTMASK
 INTUNMASK
 INTVHRS\$
 INTVMINS\$
 ISTR\$
 IVAL

J

K

L

LAUNCH
 LCTIME
 LEFMT\$
 LEFT\$
 LEN
 LEVAL

LIMITFLEN
 LIST
 LIST0
 LISTW
 LISTW0
 LITEM\$
 LITEM
 LITEMEXISTS%
 LN
 LOG
 LOG10
 LOG2
 LOGB
 LOGC
 LOGG
 LOGM
 LOGR
 LOGY
 LOHIFMT\$
 LOHIVAL
 LRC\$
 LRCHILO
 LRCLOHI

M

MASK
 MASKDRVOBJ
 MAXLITEM
 MAXMITEM
 MENU
 MENU0
 MID\$
 MINS
 MITEM\$
 MITEM
 MITEMEXISTS%
 MITEQ\$
 MKTIME
 MOD
 MON\$
 MON
 MONAB\$
 MSGENABLED%

N

NETUP%
 NEXT
 NOT
 NUMEDIT

O

OCTFMT\$
 OCTVAL
 OFFLOGSTR\$
 ON
 ONLOGSTR\$
 OPEN
 OR

P

PARAM\$
 PARAM%
 PARAM
 PARSEDEC
 PARSEREGEX
 PI

POS
 PRGNAME\$
 PRINT#
 PRINT
 PRNCHKSUM\$
 PROMPT
 PVAR\$
 PVAR%
 PVAR
 PWEDIT
 PWEDIT0

Q

QUT\$

R

RDBTN
 RDGRP
 RDGRP0
 REGERR%
 REGEXEND
 REGEXPOS
 REGHIDDEN%
 REGMASK%
 REGNAME\$
 REGSTAT%
 RELEASE
 REM
 REPEAT
 RESUME
 RET\$
 RETURN
 REVERTINDRANGE
 REVERTOFFLOGSTR
 REVERTONLOGSTR
 REVERTREGNAME
 RIGHT\$
 RND
 RNDDWN
 RNDUP
 RTSERROR
 RTSPRM\$
 RTSSEND
 RUN
 RUNRMT

S

SABUSEROR
 SABUSREPLY
 SASCII
 SBEVAL
 SECS
 SENDBIN
 SENDCMD
 SENDREPLY
 SENDSTR
 SETANAMAX
 SETANAMIN
 SETANAVAL
 SETANAVALGL
 SETANAVALS
 SETANAVALSGL
 SETBSTDLY
 SETBSTVAL
 SETDIGVAL
 SETDRVOBJVAL
 SETDRVOBJVALGL
 SETDRVOBJVALS

SETDRVOBJVALSGL
 SETFPOS
 SETINDRANGE
 SETLITEM
 SETMITEM
 SETOFFLOGSTR
 SETONLOGSTR
 SETPARAM
 SETPVAR
 SETREGNAME
 SETSTRVAL
 SHILOVAL
 SIN
 SKIPDEC
 SKIPREGEX
 SLEVAL
 SLIST
 SLIST0
 SLISTW
 SLISTW0
 SLOHIVAL
 SQRT
 STARTNET
 STEP
 STOPNET
 STR\$
 STREDIT
 STREDIT0
 STRVAL\$
 SUSPEND
 SUSPENDED%

T

TAB\$
 TAN
 THEN
 TIME\$
 TIME
 TO
 TRIGGER\$
 TRIGGERDRV\$
 TRIGGERMSG\$
 TRIGGERPRM\$
 TRIGGERPRM%
 TRIGGERPRM
 TRUE%

U

UNHIDE
 UNMASK
 UNMASKDRVOBJ
 UNTIL
 USR\$
 USRLVL
 USRPRV%

V

VAL

W

WHILE
 WKDAY\$
 WKDAY
 WKDAYAB\$

X

XOR

Y

YR

Z

Appendix B: List of Error Messages

- **Array has wrong number of dimensions**

The array you specified does not have the required number of dimensions (indices).

- **Array name expected**

The name of an array variable was expected but not found.

- **Array subscript expected**

An array variable was not followed by a subscript list.

- **Array too large**

The array has too many elements.

- **Assignment operator expected**

An assignment was missing the = sign.

- **Bad C-style string**

The string you specified contained an invalid or incomplete backslash (“\”) escape sequence.

- **Bad expression list termination**

The expression list for an INFO, ERRMSG, LOG, LOGR, LOGG, LOGB, LOGC, LOGY, LOGM, FILELOG, FILELOGR, FILELOGG, FILELOGB, FILELOGC, FILELOGY, FILELOGM, SABUSREPLY, SABUSERROR, RTSEND or RTSERROR command was terminated with a semicolon or a comma.

- **Bad hex value string**

The string you specified is not a series of double-digit hex numbers separated by spaces.

- **Bad regular expression**

The string you specified is not a valid regular expression. Note that in some instances, regular expressions that match an empty string are not valid.

- **Binary operator expected**

A binary operator in an expression is missing.

- **Boolean decoder expected**

The decoder number of a Boolean decoder was expected, but you specified the number of a string or numerical decoder.

- **Boolean expected**

A Boolean variable was expected, but a numerical or string variable was found.

- **Boolean Variable expected**

A Boolean variable was expected, but a numerical or string variable was found.

- **Byte (-128 to 127 or 0 to 255) expected**

A byte value was expected, but a string or Boolean value or a numerical value that was not a byte value was found.

- **Command or assignment expected**

A command or an assignment was expected but not found.

- **Device driver object is wrong type**

The specified data object is of the wrong type (e.g. you specified a string data object where an analog data object was required.).

- **Device driver parameter is wrong type**

The device driver parameter you specified does not have the type the function expects.

- **Dialog list has no item with that number**

The dialog list associated with the variable you specified does not have an item with the specified index.

- **Dialog list item command or function outside dialog subroutine**

You tried to access or change a dialog list's items outside the dialog's callback subroutine.

- **Dialog list item command or function variable not used in dialog**

A variable associated with a dialog list was expected, but the variable you specified is not associated with any dialog items.

- **Dialog menu has no item with that number**

The dialog menu associated with the variable you specified does not have an item with the specified index.

- **Dialog menu item command or function outside dialog subroutine**

You tried to access or change a dialog menu's items outside the dialog's callback subroutine.

- **Dialog menu item command or function variable not used in dialog**

A variable associated with a dialog menu was expected, but the variable you specified is not associated with any dialog items.

- **Dialog object is not a list**

A variable associated with a dialog list was expected, but the variable you specified is associated with a dialog item that is not a list.

- **Dialog object is not a menu**

A variable associated with a dialog menu was expected, but the variable you specified is associated with a dialog item that is not a menu.

- **Digit expected**

A number contained a non-digit.

- **Division by zero**

An operation required a division by zero.

- **"DLGERROR" outside dialog subroutine**

The `DLGERROR` command was found outside a dialog callback subroutine.

- **"DLGERROR" variable not used in dialog**

The variable specified in an `DLGERROR` command was not associated with any dialog item.

- **"DO" without "WHILE"**

The `DO` keyword was found outside of a `WHILE` statement.

- **Duplicate "ELSE"**

more than one `ELSE` command was used in a single `IF-THEN-ENDIF` block.

- **Duplicate file number**
A file with the specified file number is already open.
- **Duplicate line number**
A line with the specified line number already exists.
- **Duplicate port**
The same serial port was specified more than once in a `GRAB` command.
- **Duplicate program argument**
Two arguments with the same name were specified in a `CALL` or `RUN` command.
- **"ELSE" without "IF"**
The `ELSE` keyword was found outside of an `IF-THEN-ENDIF` block.
- **"ENDDO" without "DO"**
The `ENDDO` keyword was found without a previous `WHILE` statement.
- **"ENDIF" without "IF"**
The `ENDIF` keyword was found without a previous `IF` statement.
- **Expression is not a variable**
The name of a variable was expected, but an expression was found.
- **Expression is not an array**
The name of an array was expected, but an expression was found.
- **File not open**
The file number you specified not correspond to any open file or network connection.
- **File was opened for reading only**
You tried to write to a file that was opened for reading only.
- **File was opened for writing only**
You tried to read from a file that was opened for writing only.
- **Identifier is not a function**
Function parameters were specified for an identifier that is not a function.
- **Identifier is not an array**
An array subscript was specified for an identifier that is not an array.
- **Illegal array subscript**
The value specified as an array index was out of range, or contained a decimal point.
- **Illegal character**
A character was encountered that is not used in SCL.
- **Illegal command for device driver program**
You tried to use a command that accesses a register or a legacy parameter in a device driver program.
- **Illegal digit**
A digit other than 0 and 1 was used in a binary number, or the digit 8 or 9 was used in an octal number.

- **Illegal function for device driver program**

You tried to use a function that accesses a register or a legacy parameter in a device driver program.

- **Illegal line number**

The line number was out of range, or contained a decimal point.

- **Illegal value**

The value specified can not be used for the required purpose.

- **Inappropriate command**

A command was encountered that is not available for the use of the SCL program (e.g. you used `PRINT` in a program for a summary source.).

- **Inappropriate function**

A function was encountered that is not available for the use of the SCL program (e.g. you used `RTSPARAM$` but the program that is not an RTS control.).

- **"INPUT#" needs length for network connection**

You failed to specify a data length when using the `INPUT` command to get data from a network connection.

- **Integer (-2,147,483,648 to 2,147,483,647) expected**

A 32 bit integer value was expected, but a string or Boolean value or a numerical value that was not an integer was found.

- **Maximum number of instructions exceeded**

The instruction limit placed on the program has been exceeded. Programs for processor and summary sources are limited to 500 instruction unless otherwise specified.

- **Misplaced array subscript**

A left square bracket was found where it is not appropriate.

- **Misplaced binary operator**

A binary operator (+, -, *, /, ^, AND, OR, XOR) was found where it is not appropriate.

- **Misplaced command**

A command was found where it is not appropriate.

- **Misplaced command separator**

A colon was found where it is not appropriate.

- **Misplaced decimal point**

A decimal point was used in a hexadecimal or binary number.

- **Misplaced "ELSE"**

An `ELSE` command was found in a single command `IF-THEN` statement or a single command `WHILE-DO` loop.

- **Misplaced "ELSEIF"**

An `ELSEIF` command was found in a single command `IF-THEN` statement or a single command `WHILE-DO` loop.

- **Misplaced "ENDDO"**

An `ENDDO` command was found in a single command `IF-THEN` statement or a single command `WHILE-DO` loop.

- **Misplaced "ENDIF"**
An `ENDIF` command was found in a single command `IF-THEN` statement or a single command `WHILE-DO` loop.
- **Misplaced exponent**
An exponent was used in a binary number.
- **Misplaced line continuation character ("\\")**
A line continuation character was found in the middle of a line.
- **Misplaced "LITEM"**
An `LITEM` command was found without a previous dialog list command.
- **Misplaced "MITEM"**
An `MITEM` command was found without a previous dialog menu command.
- **Misplaced parameter separator (",")**
A comma was found where it is not appropriate.
- **Misplaced print list separator (";")**
A semicolon was found outside of an expression list.
- **Misplaced "RDBTN"**
An `RDBTN` command was found without a previous dialog radio button command.
- **Misplaced remark**
A `REM` keyword was found where it is not appropriate.
- **Misplaced sign**
A plus or minus sign was found where it is not appropriate.
- **Misplaced "UNTIL"**
An `UNTIL` command was found in a single command `IF-THEN` statement or a single command `WHILE-DO` loop.
- **Misplaced value**
A value or expression was encountered where it is not appropriate.
- **Missing array name**
A parameter list was encountered, but no array name.
- **Missing "DO"**
The `DO` keyword in a `WHILE` command was not found.
- **Missing "ENDDO"**
A `WHILE-DO` loop was not terminated by an `ENDDO`.
- **Missing "ENDIF"**
An `IF-THEN-ENDIF` block was not terminated by an `ENDIF`.
- **Missing "GOTO" or "GOSUB"**
The `GOTO` or `GOSUB` keyword in an `ON...GOTO` or `ON...GOSUB` command was not found.
- **Missing "THEN"**
The `THEN` keyword in an `IF` or `ELSEIF` command was not found.

- **Missing "TO"**

The `TO` keyword in a `FOR` command was not found.

- **"NEXT" without "FOR"**

The `NEXT` keyword was found without a corresponding `FOR` statement.

- **Network connection has no size or position**

You tried to access the size or position of a file that was really a network connection.

- **Number expected**

A numeric value or expression was expected, but a string or Boolean expression was found.

- **Numerical decoder expected**

The decoder number of a numerical decoder was expected, but you specified the number of a string or Boolean decoder.

- **Numerical variable expected**

A numerical variable was expected, but a string or Boolean variable was found.

- **Overflow**

An operation resulted in a value that is too large to be handled by SCL.

- **Parameter list expected**

A function call was missing its parameter list.

- **Port not grabbed**

The program has requested exclusive access to a number of serial ports and is attempting to access a port other than the ones it requested.

- **Positive integer (1 to 4,294,967,295) expected**

A positive unsigned 32 bit integer value was expected, but a string or Boolean value or a numerical value that was not a positive unsigned integer was found.

- **Register is wrong type**

The specified register is of the wrong register type (e.g. you specified a string register where an analog register was required.).

- **"RELEASE" without "GRAB"**

The `RELEASE` keyword was found but no ports have been grabbed.

- **"RETURN" without "GOSUB"**

The `RETURN` keyword was found outside of a subroutine.

- **Second "CANCELBTN"**

A `CANCELBTN` command was encountered, but the dialog currently being constructed already has a cancel button.

- **Second "GRAB"**

A `GRAB` command was encountered, but the program already has exclusive access to some serial ports because of a `GRAB` statement in this program, or in a program that called this program using the `CALL` command.

- **Second "SABUSREPLY" or "SABUSERROR"**

An `SABUSREPLY` or `SABUSERROR` command was encountered, but a reply has already already been sent using one of those two commands.

- **Single statement after "ELSEIF"**

An `ELSIF` command was followed by a single command or assignment. `ELSEIF` must be followed by a block of code.

- **"STEP" without "FOR"**

The `STEP` keyword was found outside of a `FOR` statement.

- **String contains non-printable characters**

A string containing non-printable characters (ASCII \$00-\$1F or \$7F to \$FF) was encountered in an `SABUSREPLY` or `SABUSEROR` command.

- **String decoder expected**

The decoder number of a string decoder was expected, but you specified the number of a numerical or Boolean decoder.

- **String expected**

A string value or expression was expected, but a numerical or Boolean expression was found.

- **String Variable expected**

A string variable was expected, but a numerical or Boolean variable was found.

- **Superfluous array subscript**

You specified a subscript for an array where you should only have specified the array's name.

- **Syntax error**

The syntax of a statement was incomprehensible.

- **"THEN" without "IF"**

The `THEN` keyword was found outside of an `IF` statement.

- **"TO" without "FOR"**

The `TO` keyword was found outside of a `FOR` statement.

- **Too few arguments**

The command requires more arguments.

- **Too few indices**

The array requires more indices.

- **Too few parameters**

The function requires more parameters.

- **Too many arguments**

The command does not allow that many arguments.

- **Too many indices**

The array does not require that many indices.

- **Too many parameters**

The function does not take that many parameters.

- **Trigger object parameter is wrong type**

The trigger object parameter you specified does not have the type the function expects.

- **Type mismatch**

Two types (numeric, string, Boolean) that are required to match do not match.

- **Unexpected end of line**

The line of code is incomplete.

- **Unknown application**

The specified application could not be found, or could not be launched for other reasons.

- **Unknown decoder**

No data decoder with the specified decoder number exists.

- **Unknown device command**

The device driver does not contain a command with the specified tag.

- **Unknown device driver**

No device driver with the specified tag exists.

- **Unknown device driver object**

The device driver does not contain a data object with the specified tag.

- **Unknown device driver parameter**

The device driver the program belongs to does not have a parameter with the specified tag.

- **Unknown device message**

The device driver does not contain a legacy device message with the specified tag.

- **Unknown device reply**

The device driver does not contain a legacy device reply with the specified tag.

- **Unknown device response**

The device driver does not contain a legacy device response with the specified tag.

- **Unknown encoder**

No data encoder with the specified encoder number exists.

- **Unknown line number**

The specified line number does not exist.

- **Unknown parameter**

No legacy parameter with the specified tag exists.

- **Unknown register**

No register with the specified tag exists.

- **Unknown SCL program**

No SCL program with the specified tag exists.

- **Unknown serial port**

No serial port with the specified tag exists.

- **Unknown trigger object parameter**

The trigger of the program does not have a parameter with the specified tag.

- **Unmatched left parenthesis ("(")**

A left parenthesis without a corresponding right parenthesis was encountered.

- **Unmatched right parenthesis (")")**

A right parenthesis without a corresponding left parenthesis was encountered.

- **Unmatched start array subscript character ("[")**
The right square bracket of an array subscript list was missing.
- **Unmatched string delimiter (double quotes)**
The closing quotes for the string are missing.
- **Unsigned integer (0 to 4,294,967,295) expected**
An unsigned 32 bit integer value was expected, but a string or Boolean value or a numerical value that was not an unsigned integer was found.
- **"UNTIL" without "REPEAT"**
The `UNTIL` keyword was found without a previous `REPEAT` statement.
- **Value expected**
A value was expected but not found.
- **Variable is not an array**
An array name was expected, but the variable you specified is not an array.
- **Variable name expected**
A variable name was expected but not found.
- **Wrong "NEXT" variable**
The variable specified in the `NEXT` statement was not the same as that specified in the last `FOR` statement.

CONTACT INFORMATION

U.P.M.A.C.S. Communications Inc.

714, 36th Ave., Suite 301
Lachine, QC, Canada
H8T 3L8

Tel: 1-514-697-5500
Toll free: 1-877-697-5500 (Canada & US only)
E-Mail: support@upmacs.com

UPMACS is on the Web at <http://www.upmacs.com>