

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**Devices and integration in enterprise systems – Function blocks (FB) for
process control and electronic device description language (EDDL) –
Part 5: EDDL Built-in library**

**Les dispositifs et leur intégration dans les systèmes de l'entreprise – blocs
fonctionnels (FB) pour les procédés industriels et le langage de description
électronique de produit (EDDL) –
Partie 5: Bibliothèque de Built-in EDDL**



THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2020 IEC, Geneva, Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either IEC or IEC's member National Committee in the country of the requester. If you have any questions about IEC copyright or have an enquiry about obtaining additional rights to this publication, please contact the address below or your local IEC member National Committee for further information.

Droits de reproduction réservés. Sauf indication contraire, aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de l'IEC ou du Comité national de l'IEC du pays du demandeur. Si vous avez des questions sur le copyright de l'IEC ou si vous désirez obtenir des droits supplémentaires sur cette publication, utilisez les coordonnées ci-après ou contactez le Comité national de l'IEC de votre pays de résidence.

IEC Central Office
3, rue de Varembe
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
info@iec.ch
www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigendum or an amendment might have been published.

IEC publications search - webstore.iec.ch/advsearchform

The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee,...). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

IEC Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: sales@iec.ch.

Electropedia - www.electropedia.org

The world's leading online dictionary on electrotechnology, containing more than 22 000 terminological entries in English and French, with equivalent terms in 16 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary

67 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

A propos de l'IEC

La Commission Electrotechnique Internationale (IEC) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications IEC

Le contenu technique des publications IEC est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Recherche de publications IEC -

webstore.iec.ch/advsearchform

La recherche avancée permet de trouver des publications IEC en utilisant différents critères (numéro de référence, texte, comité d'études,...). Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

IEC Just Published - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications IEC. Just Published détaille les nouvelles publications parues. Disponible en ligne et une fois par mois par email.

Service Clients - webstore.iec.ch/csc

Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: sales@iec.ch.

Electropedia - www.electropedia.org

Le premier dictionnaire d'électrotechnologie en ligne au monde, avec plus de 22 000 articles terminologiques en anglais et en français, ainsi que les termes équivalents dans 16 langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (IEV) en ligne.

Glossaire IEC - std.iec.ch/glossary

67 000 entrées terminologiques électrotechniques, en anglais et en français, extraites des articles Termes et Définitions des publications IEC parues depuis 2002. Plus certaines entrées antérieures extraites des publications des CE 37, 77, 86 et CISPR de l'IEC.

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**Devices and integration in enterprise systems – Function blocks (FB) for
process control and electronic device description language (EDDL) –
Part 5: EDDL Builtin library**

**Les dispositifs et leur intégration dans les systèmes de l'entreprise – blocs
fonctionnels (FB) pour les procédés industriels et le langage de description
électronique de produit (EDDL) –
Partie 5: Bibliothèque de Builtin EDDL**

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

COMMISSION
ELECTROTECHNIQUE
INTERNATIONALE

ICS 25.040.40; 35.240.50

ISBN 978-2-8322-8446-9

**Warning! Make sure that you obtained this publication from an authorized distributor.
Attention! Veuillez vous assurer que vous avez obtenu cette publication via un distributeur agréé.**

CONTENTS

FOREWORD.....	20
INTRODUCTION.....	23
1 Scope.....	24
2 Normative references	24
3 Terms, definitions, acronyms and abbreviated terms	24
3.1 Terms and definitions.....	24
3.2 Acronyms and abbreviated terms	24
4 Conventions for builtin descriptions	25
5 Builtin categories.....	26
5.1 Overview.....	26
5.2 User interface builtins	36
5.3 Communication builtins	37
5.4 Action builtins	39
6 EDDL builtin library.....	42
6.1 General.....	42
6.2 User interface builtins	42
6.2.1 Builtin ACKNOWLEDGE	42
6.2.2 Builtin acknowledge	43
6.2.3 Builtin DELAY	44
6.2.4 Builtin delay.....	44
6.2.5 Builtin delayfor.....	45
6.2.6 Builtin delayfor2.....	46
6.2.7 Builtin DISPLAY.....	48
6.2.8 Builtin display	49
6.2.9 Builtin display_bitenum	49
6.2.10 Builtin display_builtin_error.....	50
6.2.11 Builtin display_comm_error.....	50
6.2.12 Builtin display_comm_status.....	51
6.2.13 Builtin display_device_status	52
6.2.14 Builtin display_dynamics.....	52
6.2.15 Builtin display_dynamics2.....	53
6.2.16 Builtin display_message	54
6.2.17 Builtin display_message2	55
6.2.18 Builtin display_response_code.....	57
6.2.19 Builtin display_response_status.....	58
6.2.20 Builtin display_xmtr_status	58
6.2.21 Builtin edit_device_value	59
6.2.22 Builtin edit_device_value2	60
6.2.23 Builtin edit_local_value	62
6.2.24 Builtin edit_local_value2	63
6.2.25 Builtin get_acknowledgement	64
6.2.26 Builtin get_acknowledgement2.....	65
6.2.27 Builtin GET_DEV_VAR_VALUE	66
6.2.28 Builtin get_dev_var_value.....	67
6.2.29 Builtin GET_LOCAL_VAR_VALUE	68
6.2.30 Builtin get_local_var_value	68

6.2.31	Builtin Menu	69
6.2.32	Builtin MenuDisplay (version A)	70
6.2.33	Builtin MenuDisplay (version B)	71
6.2.34	Builtin PUT_MESSAGE	72
6.2.35	Builtin put_message	73
6.2.36	Builtin SELECT_FROM_LIST	75
6.2.37	Builtin select_from_list	76
6.2.38	Builtin select_from_menu	76
6.2.39	Builtin select_from_menu2	78
6.2.40	Prompt string formats	79
6.3	Communication builtins	80
6.3.1	Builtin abort_on_all_comm_errors	80
6.3.2	Builtin ABORT_ON_ALL_COMM_STATUS	81
6.3.3	Builtin ABORT_ON_ALL_DEVICE_STATUS	81
6.3.4	Builtin ABORT_ON_ALL_RESPONSE_CODES	82
6.3.5	Builtin abort_on_all_response_codes	82
6.3.6	Builtin ABORT_ON_COMM_ERROR	83
6.3.7	Builtin abort_on_comm_error	83
6.3.8	Builtin ABORT_ON_COMM_STATUS	84
6.3.9	Builtin ABORT_ON_DEVICE_STATUS	85
6.3.10	Builtin ABORT_ON_NO_DEVICE (deprecated)	85
6.3.11	Builtin ABORT_ON_RESPONSE_CODE	86
6.3.12	Builtin abort_on_response_code	87
6.3.13	Builtin ext_send_command	87
6.3.14	Builtin ext_send_command_trans	88
6.3.15	Builtin fail_on_all_comm_errors	89
6.3.16	Builtin fail_on_all_response_codes	90
6.3.17	Builtin fail_on_comm_error	90
6.3.18	Builtin fail_on_response_code	91
6.3.19	Builtin get_more_status	92
6.3.20	Builtin IGNORE_ALL_COMM_STATUS	92
6.3.21	Builtin IGNORE_ALL_DEVICE_STATUS	93
6.3.22	Builtin IGNORE_ALL_RESPONSE_CODES	93
6.3.23	Builtin IGNORE_COMM_ERROR	94
6.3.24	Builtin IGNORE_COMM_STATUS	95
6.3.25	Builtin IGNORE_DEVICE_STATUS	95
6.3.26	Builtin IGNORE_NO_DEVICE (deprecated)	96
6.3.27	Builtin IGNORE_RESPONSE_CODE	96
6.3.28	Builtin read_value	97
6.3.29	Builtin read_value2	97
6.3.30	Builtin ReadCommand	98
6.3.31	Builtin retry_on_all_comm_errors	99
6.3.32	Builtin RETRY_ON_ALL_COMM_STATUS	99
6.3.33	Builtin RETRY_ON_ALL_DEVICE_STATUS	100
6.3.34	Builtin RETRY_ON_ALL_RESPONSE_CODES	100
6.3.35	Builtin retry_on_all_response_codes	101
6.3.36	Builtin RETRY_ON_COMM_ERROR	102
6.3.37	Builtin retry_on_comm_error	102
6.3.38	Builtin RETRY_ON_COMM_STATUS	103

6.3.39	Builtin RETRY_ON_DEVICE_STATUS.....	103
6.3.40	Builtin RETRY_ON_NO_DEVICE (deprecated)	104
6.3.41	Builtin RETRY_ON_RESPONSE_CODE	105
6.3.42	Builtin retry_on_response_code.....	105
6.3.43	Builtin send.....	106
6.3.44	Builtin send_all_values	107
6.3.45	Builtin send_command.....	107
6.3.46	Builtin send_command_trans	108
6.3.47	Builtin send_trans	108
6.3.48	Builtin send_value	109
6.3.49	Builtin send_value2	110
6.3.50	Builtin SET_NUMBER_OF_RETRIES	111
6.3.51	Builtin WriteCommand	111
6.3.52	Builtin XMTR_ABORT_ON_ALL_COMM_STATUS.....	112
6.3.53	Builtin XMTR_ABORT_ON_ALL_DATA.....	112
6.3.54	Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS.....	113
6.3.55	Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES	113
6.3.56	Builtin XMTR_ABORT_ON_COMM_ERROR	114
6.3.57	Builtin XMTR_ABORT_ON_COMM_STATUS	114
6.3.58	Builtin XMTR_ABORT_ON_DATA.....	115
6.3.59	Builtin XMTR_ABORT_ON_DEVICE_STATUS.....	116
6.3.60	Builtin XMTR_ABORT_ON_NO_DEVICE (deprecated)	116
6.3.61	Builtin XMTR_ABORT_ON_RESPONSE_CODE	117
6.3.62	Builtin XMTR_IGNORE_ALL_COMM_STATUS	117
6.3.63	Builtin XMTR_IGNORE_ALL_DATA	118
6.3.64	Builtin XMTR_IGNORE_ALL_DEVICE_STATUS	118
6.3.65	Builtin XMTR_IGNORE_ALL_RESPONSE_CODES	119
6.3.66	Builtin XMTR_IGNORE_COMM_ERROR	119
6.3.67	Builtin XMTR_IGNORE_COMM_STATUS	120
6.3.68	Builtin XMTR_IGNORE_DATA	120
6.3.69	Builtin XMTR_IGNORE_DEVICE_STATUS	121
6.3.70	Builtin XMTR_IGNORE_NO_DEVICE (deprecated).....	121
6.3.71	Builtin XMTR_IGNORE_RESPONSE_CODE.....	122
6.3.72	Builtin XMTR_RETRY_ON_ALL_COMM_STATUS	123
6.3.73	Builtin XMTR_RETRY_ON_ALL_DATA	123
6.3.74	Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS	124
6.3.75	Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES	124
6.3.76	Builtin XMTR_RETRY_ON_COMM_ERROR	125
6.3.77	Builtin XMTR_RETRY_ON_COMM_STATUS	125
6.3.78	Builtin XMTR_RETRY_ON_DATA	126
6.3.79	Builtin XMTR_RETRY_ON_DEVICE_STATUS	126
6.3.80	Builtin XMTR_RETRY_ON_NO_DEVICE (deprecated).....	127
6.3.81	Builtin XMTR_RETRY_ON_RESPONSE_CODE.....	127
6.4	Action builtins	128
6.4.1	Builtin fgetval.....	128
6.4.2	Builtin fsetval	128
6.4.3	Builtin get_date	129
6.4.4	Builtin get_double.....	130
6.4.5	Builtin get_float.....	130

6.4.6	Builtin get_signed	131
6.4.7	Builtin get_string.....	132
6.4.8	Builtin get_unsigned	132
6.4.9	Builtin igetval.....	133
6.4.10	Builtin isetval	133
6.4.11	Builtin isOffline	134
6.4.12	Builtin lgetval	134
6.4.13	Builtin lsetval	135
6.4.14	Builtin put_date	135
6.4.15	Builtin put_double.....	136
6.4.16	Builtin put_float.....	137
6.4.17	Builtin put_signed	137
6.4.18	Builtin put_string.....	138
6.4.19	Builtin put_unsigned	139
6.4.20	Builtin sgetval	139
6.4.21	Builtin ssetval	140
6.5	Variable access builtins	140
6.5.1	Builtin assign	140
6.5.2	Builtin assign_double.....	141
6.5.3	Builtin assign_float	142
6.5.4	Builtin assign_int	142
6.5.5	Builtin assign_var	143
6.5.6	Builtin assign2	143
6.5.7	Builtin dassign	145
6.5.8	Builtin fassign	145
6.5.9	Builtin float_value	145
6.5.10	Builtin fvar_value	146
6.5.11	Builtin get_date_value	146
6.5.12	Builtin get_date_value2	147
6.5.13	Builtin get_double_value.....	148
6.5.14	Builtin get_double_value2.....	149
6.5.15	Builtin get_float_value	150
6.5.16	Builtin get_float_value2	151
6.5.17	Builtin get_signed_value	152
6.5.18	Builtin get_signed_value2.....	152
6.5.19	Builtin get_string_value	153
6.5.20	Builtin get_string_value2.....	154
6.5.21	Builtin get_unsigned_value	155
6.5.22	Builtin get_unsigned_value2	156
6.5.23	Builtin iassign	157
6.5.24	Builtin int_value	157
6.5.25	Builtin ivar_value	158
6.5.26	Builtin lassign	158
6.5.27	Builtin long_value	159
6.5.28	Builtin lvar_value	159
6.5.29	Builtin put_date_value	159
6.5.30	Builtin put_date_value2	160
6.5.31	Builtin put_double_value.....	161
6.5.32	Builtin put_double_value2.....	162

6.5.33	Builtin put_float_value	163
6.5.34	Builtin put_float_value2	164
6.5.35	Builtin put_signed_value	165
6.5.36	Builtin put_signed_value2	165
6.5.37	Builtin put_string_value	166
6.5.38	Builtin put_string_value2	167
6.5.39	Builtin put_unsigned_value	168
6.5.40	Builtin put_unsigned_value2	169
6.5.41	Builtin ret_double_value	170
6.5.42	Builtin ret_double_value2	171
6.5.43	Builtin ret_float_value	171
6.5.44	Builtin ret_float_value2	172
6.5.45	Builtin ret_signed_value	172
6.5.46	Builtin ret_signed_value2	173
6.5.47	Builtin ret_unsigned_value	174
6.5.48	Builtin ret_unsigned_value2	174
6.5.49	Builtin vassign	175
6.6	Normal termination builtins	175
6.6.1	Builtin discard_on_exit	175
6.6.2	Builtin save_on_exit	176
6.6.3	Builtin save_values	177
6.6.4	Builtin send_on_exit	178
6.7	Abnormal termination builtins	179
6.7.1	Builtin abort	179
6.7.2	Builtin add_abort_method (version A)	179
6.7.3	Builtin add_abort_method (version B)	180
6.7.4	Builtin method_abort	180
6.7.5	Builtin pop_abort_method	181
6.7.6	Builtin process_abort	181
6.7.7	Builtin push_abort_method	182
6.7.8	Builtin remove_abort_method (version A)	182
6.7.9	Builtin remove_abort_method (version B)	183
6.7.10	Builtin remove_all_abort_methods	183
6.8	String builtins	184
6.8.1	Builtin atof	184
6.8.2	Builtin atoi	184
6.8.3	Builtin BUILD_MESSAGE	185
6.8.4	Builtin dictionary_string	185
6.8.5	Builtin ftoa	186
6.8.6	Builtin get_comm_error_string	186
6.8.7	Builtin get_dictionary_string	187
6.8.8	Builtin get_enum_string	188
6.8.9	Builtin get_response_code_string	188
6.8.10	Builtin get_rspcode_string	189
6.8.11	Builtin get_rspcode_string_by_id	190
6.8.12	Builtin get_status_code_string (deprecated)	190
6.8.13	Builtin get_status_string	191
6.8.14	Builtin get_stddict_string	192
6.8.15	Builtin get_variable_string	192

6.8.16	Builtin itoa (version A)	193
6.8.17	Builtin itoa (version B)	193
6.8.18	Builtin strleft	194
6.8.19	Builtin strcmp	194
6.8.20	Builtin strlen	195
6.8.21	Builtin strlwr	196
6.8.22	Builtin strmid	196
6.8.23	Builtin strright	197
6.8.24	Builtin strstr	197
6.8.25	Builtin strtrim	198
6.8.26	Builtin strupr	199
6.8.27	Builtin trunc	199
6.9	Date time builtins	200
6.9.1	Builtin AddTime	200
6.9.2	Builtin DATE_AND_TIME_VALUE_to_string	200
6.9.3	Builtin Date_to_DayOfMonth	201
6.9.4	Builtin DATE_to_days	201
6.9.5	Builtin Date_to_Month	202
6.9.6	Builtin DATE_to_string	202
6.9.7	Builtin Date_To_Time	203
6.9.8	Builtin Date_to_Year	203
6.9.9	Builtin days_to_DATE	204
6.9.10	Builtin DELAY_TIME	204
6.9.11	Builtin DiffTime	204
6.9.12	Builtin From_DATE_AND_TIME_VALUE	205
6.9.13	Builtin From_TIME_VALUE	205
6.9.14	Builtin GetCurrentDate	206
6.9.15	Builtin GetCurrentDateAndTime	206
6.9.16	Builtin GetCurrentTime	207
6.9.17	Builtin GET_TICK_COUNT	207
6.9.18	Builtin Make_Time	207
6.9.19	Builtin seconds_to_TIME_VALUE	208
6.9.20	Builtin seconds_to_TIME_VALUE8	209
6.9.21	Builtin Time_To_Date	209
6.9.22	Builtin TIME_VALUE_to_Hour	209
6.9.23	Builtin TIME_VALUE_to_Minute	210
6.9.24	Builtin TIME_VALUE_to_Second	210
6.9.25	Builtin TIME_VALUE_to_seconds	211
6.9.26	Builtin TIME_VALUE_to_string	211
6.9.27	Builtin timet_to_string	212
6.9.28	Builtin timet_to_TIME_VALUE	213
6.9.29	Builtin timet_To_TIME_VALUE	213
6.9.30	Builtin timet_to_TIME_VALUE8	214
6.9.31	Builtin To_Date	214
6.9.32	Builtin To_Date_and_Time	215
6.9.33	Builtin To_Time	215
6.9.34	Builtin To_TIME_VALUE	216
6.9.35	Builtin To_TIME_VALUE8	216
6.10	Diagnostic builtins	217

6.10.1	Builtin get_comm_error	217
6.10.2	Builtin get_dds_error	217
6.10.3	Builtin get_response_code	218
6.11	Math builtins	219
6.11.1	Builtin abs	219
6.11.2	Builtin acos	219
6.11.3	Builtin asin	220
6.11.4	Builtin atan	220
6.11.5	Builtin ByteToDouble	221
6.11.6	Builtin ByteToFloat	221
6.11.7	Builtin ByteToLong	222
6.11.8	Builtin ByteToShort	222
6.11.9	Builtin cbrt	223
6.11.10	Builtin ceil	223
6.11.11	Builtin cos	224
6.11.12	Builtin cosh	224
6.11.13	Builtin DoubleToByte	225
6.11.14	Builtin drand	225
6.11.15	Builtin dseed	226
6.11.16	Builtin exp	226
6.11.17	Builtin FloatToByte	226
6.11.18	Builtin floor	227
6.11.19	Builtin fmod	227
6.11.20	Builtin fpclassify	228
6.11.21	Builtin is_NaN	229
6.11.22	Builtin log	229
6.11.23	Builtin log10	229
6.11.24	Builtin log2	230
6.11.25	Builtin LongToByte	230
6.11.26	Builtin nan	231
6.11.27	Builtin NaN_value	232
6.11.28	Builtin nanf	232
6.11.29	Builtin pow	233
6.11.30	Builtin round	233
6.11.31	Builtin ShortToByte	234
6.11.32	Builtin sin	234
6.11.33	Builtin sinh	235
6.11.34	Builtin sqrt	235
6.11.35	Builtin tan	235
6.11.36	Builtin tanh	236
6.12	List builtins	236
6.12.1	Builtin get_date_lelem	236
6.12.2	Builtin get_date_lelem2	237
6.12.3	Builtin get_double_lelem	238
6.12.4	Builtin get_double_lelem2	239
6.12.5	Builtin get_float_lelem	240
6.12.6	Builtin get_float_lelem2	241
6.12.7	Builtin get_signed_lelem	242
6.12.8	Builtin get_signed_lelem2	242

6.12.9	Builtin get_string_lelem	243
6.12.10	Builtin get_string_lelem2.....	244
6.12.11	Builtin get_unsigned_lelem	245
6.12.12	Builtin get_unsigned_lelem2	246
6.12.13	Builtin ListDeleteElementAt (version A).....	247
6.12.14	Builtin ListDeleteElementAt (version B).....	248
6.12.15	Builtin ListDeleteElementAt2.....	248
6.12.16	Builtin ListInsert (version A).....	249
6.12.17	Builtin ListInsert (version B).....	250
6.12.18	Builtin ListInsert2.....	250
6.13	Translation builtins.....	251
6.13.1	Builtin DICT_ID.....	251
6.13.2	Builtin get_block_instance_by_object_index	252
6.13.3	Builtin get_block_instance_by_tag	253
6.13.4	Builtin get_block_instance_count.....	254
6.13.5	Builtin get_resolve_status.....	254
6.13.6	Builtin ITEM_ID	255
6.13.7	Builtin MEMBER_ID	255
6.13.8	Builtin ObjectReference	256
6.13.9	Builtin resolve_array_ref.....	257
6.13.10	Builtin resolve_array_ref2.....	258
6.13.11	Builtin resolve_block_ref.....	258
6.13.12	Builtin resolve_block_ref2.....	259
6.13.13	Builtin resolve_list_ref	260
6.13.14	Builtin resolve_local_ref.....	260
6.13.15	Builtin resolve_local_ref2.....	261
6.13.16	Builtin resolve_param_list_ref.....	262
6.13.17	Builtin resolve_param_ref	262
6.13.18	Builtin resolve_param_ref2	263
6.13.19	Builtin resolve_record_ref.....	264
6.13.20	Builtin resolve_record_ref2	264
6.13.21	Builtin VARID.....	265
6.14	Block transfer builtins	265
6.14.1	Builtin abortTransferPort.....	265
6.14.2	Builtin closeTransferPort.....	266
6.14.3	Builtin fGetByte	266
6.14.4	Builtin get_transfer_status	267
6.14.5	Builtin openTransferPort.....	268
6.14.6	Builtin readItemFromDevice.....	268
6.14.7	Builtin writeItemToDevice	269
6.15	File builtins	269
6.15.1	Builtin browseIdentity.....	269
6.15.2	Builtin re_read_file.....	270
6.15.3	Builtin re_write_file	270
6.16	Identification builtins	271
6.16.1	Builtin GET_DD_REVISION	271
6.16.2	Builtin GET_DEVICE_REVISION	271
6.16.3	Builtin GET_DEVICE_TYPE.....	271
6.16.4	Builtin GET_MANUFACTURER.....	272

6.17	Development support builtins	272
6.17.1	Builtin _ERROR	272
6.17.2	Builtin _TRACE	273
6.17.3	Builtin _WARNING	273
6.17.4	Builtin LOG_MESSAGE	274
6.18	DDL references	274
6.18.1	Direct referencing of VARIABLES	274
6.18.2	Accessing standard dictionary strings	275
6.18.3	String support using DD_STRING	276
6.18.4	Referencing DD item attributes	276
6.19	Method functions	277
7	Builtins return codes	278
Figure 1 – A sample METHOD		278
Figure 2 – DD_STRING parameters passed by reference		278
Table 1 – Format for the builtin overview table		25
Table 2 – Contents of the builtin overview table		25
Table 3 – Format for the builtins lexical element tables		25
Table 4 – Contents of the lexical element table		26
Table 5 – Builtin overview		26
Table 6 – Usage of builtins		36
Table 7 – User interface builtins		36
Table 8 – Communication builtins		37
Table 9 – Action builtins		39
Table 10 – Builtin ACKNOWLEDGE		43
Table 11 – Builtin acknowledge		43
Table 12 – Builtin DELAY		44
Table 13 – Builtin delay		45
Table 14 – Builtin delayfor		45
Table 15 – Builtin delayfor2		47
Table 16 – Builtin DISPLAY		48
Table 17 – Builtin display		49
Table 18 – Builtin display_bitenum		49
Table 19 – Builtin display_builtin_error		50
Table 20 – Builtin display_comm_error		51
Table 21 – Builtin display_comm_status		51
Table 22 – Builtin display_device_status		52
Table 23 – Builtin display_dynamics		53
Table 24 – Builtin display_dynamics2		54
Table 25 – Builtin display_message		55
Table 26 – Builtin display_message2		56
Table 27 – Builtin display_response_code		57
Table 28 – Builtin display_response_status		58

Table 29 – Builtin display_xmtr_status	59
Table 30 – Builtin edit_device_value	59
Table 31 – Builtin edit_device_value2	61
Table 32 – Builtin edit_local_value	62
Table 33 – Builtin edit_local_value2	64
Table 34 – Builtin get_acknowledgement	65
Table 35 – Builtin get_acknowledgement2	66
Table 36 – Builtin GET_DEV_VAR_VALUE	67
Table 37 – Builtin get_dev_var_value	68
Table 38 – Builtin GET_LOCAL_VAR_VALUE	68
Table 39 – Builtin get_local_var_value	69
Table 40 – Builtin Menu	70
Table 41 – Builtin MenuDisplay	71
Table 42 – Builtin MenuDisplay	72
Table 43 – Builtin PUT_MESSAGE	73
Table 44 – Builtin put_message	74
Table 45 – Builtin SELECT_FROM_LIST	75
Table 46 – Builtin select_from_list	76
Table 47 – Builtin select_from_menu	77
Table 48 – Builtin select_from_menu2	79
Table 49 – Format options	80
Table 50 – Builtin abort_on_all_comm_errors	80
Table 51 – Builtin ABORT_ON_ALL_COMM_STATUS	81
Table 52 – Builtin ABORT_ON_ALL_DEVICE_STATUS	82
Table 53 – Builtin ABORT_ON_ALL_RESPONSE_CODES	82
Table 54 – Builtin abort_on_all_response_codes	83
Table 55 – Builtin ABORT_ON_COMM_ERROR	83
Table 56 – Builtin abort_on_comm_error	84
Table 57 – Builtin ABORT_ON_COMM_STATUS	84
Table 58 – Builtin ABORT_ON_DEVICE_STATUS	85
Table 59 – Builtin ABORT_ON_NO_DEVICE	86
Table 60 – Available abort and retry masks	86
Table 61 – Builtin ABORT_ON_RESPONSE_CODE	87
Table 62 – Builtin abort_on_response_code	87
Table 63 – Builtin ext_send_command	88
Table 64 – Builtin ext_send_command_trans	89
Table 65 – Builtin fail_on_all_comm_errors	90
Table 66 – Builtin fail_on_all_response_codes	90
Table 67 – Builtin fail_on_comm_error	91
Table 68 – Builtin fail_on_response_code	91
Table 69 – Builtin get_more_status	92
Table 70 – Builtin IGNORE_ALL_COMM_STATUS	93
Table 71 – Builtin IGNORE_ALL_DEVICE_STATUS	93

Table 72 – Builtin IGNORE_ALL_RESPONSE_CODES.....	94
Table 73 – Builtin IGNORE_COMM_ERROR.....	94
Table 74 – Builtin IGNORE_COMM_STATUS	95
Table 75 – Builtin IGNORE_DEVICE_STATUS	96
Table 76 – Builtin IGNORE_NO_DEVICE.....	96
Table 77 – Builtin IGNORE_RESPONSE_CODE.....	97
Table 78 – Builtin read_value.....	97
Table 79 – Builtin read_value2.....	98
Table 80 – Builtin ReadCommand	99
Table 81 – Builtin retry_on_all_comm_errors	99
Table 82 – Builtin RETRY_ON_ALL_COMM_STATUS	100
Table 83 – Builtin RETRY_ON_ALL_DEVICE_STATUS	100
Table 84 – Builtin RETRY_ON_ALL_RESPONSE_CODES.....	101
Table 85 – Builtin retry_on_all_response_codes	101
Table 86 – Builtin RETRY_ON_COMM_ERROR.....	102
Table 87 – Builtin retry_on_comm_error	103
Table 88 – Builtin RETRY_ON_COMM_STATUS	103
Table 89 – Builtin RETRY_ON_DEVICE_STATUS	104
Table 90 – Builtin RETRY_ON_NO_DEVICE.....	104
Table 91 – Builtin RETRY_ON_RESPONSE_CODE.....	105
Table 92 – Builtin retry_on_response_code	106
Table 93 – Builtin send	106
Table 94 – Builtin send_all_values.....	107
Table 95 – Builtin send_command	108
Table 96 – Builtin send_command_trans.....	108
Table 97 – Builtin send_trans.....	109
Table 98 – Builtin send_value.....	110
Table 99 – Builtin send_value2	111
Table 100 – Builtin SET_NUMBER_OF_RETRIES	111
Table 101 – Builtin WriteCommand	112
Table 102 – Builtin XMTR_ABORT_ON_ALL_COMM_STATUS.....	112
Table 103 – Builtin XMTR_ABORT_ON_ALL_DATA.....	113
Table 104 – Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS.....	113
Table 105 – Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES	114
Table 106 – Builtin XMTR_ABORT_ON_COMM_ERROR.....	114
Table 107 – Builtin XMTR_ABORT_ON_COMM_STATUS.....	115
Table 108 – Builtin XMTR_ABORT_ON_DATA.....	115
Table 109 – Builtin XMTR_ABORT_ON_DEVICE_STATUS.....	116
Table 110 – Builtin XMTR_ABORT_ON_NO_DEVICE	117
Table 111 – Builtin XMTR_ABORT_ON_RESPONSE_CODE	117
Table 112 – Builtin XMTR_IGNORE_ALL_COMM_STATUS	118
Table 113 – Builtin XMTR_IGNORE_ALL_DATA.....	118
Table 114 – Builtin XMTR_IGNORE_ALL_DEVICE_STATUS	119

Table 115 – Builtin XMTR_IGNORE_ALL_RESPONSE_CODES	119
Table 116 – Builtin XMTR_IGNORE_COMM_ERROR	120
Table 117 – Builtin XMTR_IGNORE_COMM_STATUS	120
Table 118 – Builtin XMTR_IGNORE_DATA	121
Table 119 – Builtin XMTR_IGNORE_DEVICE_STATUS	121
Table 120 – Builtin XMTR_IGNORE_NO_DEVICE	122
Table 121 – Builtin XMTR_IGNORE_RESPONSE_CODE	122
Table 122 – Builtin XMTR_RETRY_ON_ALL_COMM_STATUS	123
Table 123 – Builtin XMTR_RETRY_ON_ALL_DATA	123
Table 124 – Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS	124
Table 125 – Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES	125
Table 126 – Builtin XMTR_RETRY_ON_COMM_ERROR	125
Table 127 – Builtin XMTR_RETRY_ON_COMM_STATUS	126
Table 128 – Builtin XMTR_RETRY_ON_DATA	126
Table 129 – Builtin XMTR_RETRY_ON_DEVICE_STATUS	127
Table 130 – Builtin XMTR_RETRY_ON_NO_DEVICE	127
Table 131 – Builtin XMTR_RETRY_ON_RESPONSE_CODE	128
Table 132 – Builtin fgetval	128
Table 133 – Builtin fsetval	129
Table 134 – Builtin get_date	129
Table 135 – Builtin get_double	130
Table 136 – Builtin get_float	131
Table 137 – Builtin get_signed	131
Table 138 – Builtin get_string	132
Table 139 – Builtin get_unsigned	133
Table 140 – Builtin igetval	133
Table 141 – Builtin isetval	134
Table 142 – Builtin isOffline	134
Table 143 – Builtin lgetval	135
Table 144 – Builtin lsetval	135
Table 145 – Builtin put_date	136
Table 146 – Builtin put_double	136
Table 147 – Builtin put_float	137
Table 148 – Builtin put_signed	138
Table 149 – Builtin put_string	138
Table 150 – Builtin put_unsigned	139
Table 151 – Builtin sgetval	140
Table 152 – Builtin ssetval	140
Table 153 – Builtin assign	141
Table 154 – Builtin assign_double	141
Table 155 – Builtin assign_float	142
Table 156 – Builtin assign_int	142
Table 157 – Builtin assign_var	143

Table 158 – Builtin assign2	144
Table 159 – Builtin dassign	145
Table 160 – Builtin fassign	145
Table 161 – Builtin float_value	146
Table 162 – Builtin fvar_value	146
Table 163 – Builtin get_date_value	147
Table 164 – Builtin get_date_value2	148
Table 165 – Builtin get_double_value	149
Table 166 – Builtin get_double_value2	150
Table 167 – Builtin get_float_value	150
Table 168 – Builtin get_float_value2	151
Table 169 – builtin get_signed_value	152
Table 170 – Builtin get_signed_value2	153
Table 171 – Builtin get_string_value	154
Table 172 – Builtin get_string_value2	155
Table 173 – Builtin get_unsigned_value	156
Table 174 – Builtin get_unsigned_value2	157
Table 175 – Builtin iassign	157
Table 176 – Builtin int_value	158
Table 177 – Builtin ivar_value	158
Table 178 – Builtin lassign	158
Table 179 – Builtin long_value	159
Table 180 – Builtin lvar_value	159
Table 181 – Builtin put_date_value	160
Table 182 – Builtin put_date_value2	161
Table 183 – Builtin put_double_value	162
Table 184 – Builtin put_double_value2	163
Table 185 – Builtin put_float_value	163
Table 186 – Builtin put_float_value2	164
Table 187 – Builtin put_signed_value	165
Table 188 – Builtin put_signed_value2	166
Table 189 – Builtin put_string_value	167
Table 190 – Builtin put_string_value2	168
Table 191 – Builtin put_unsigned_value	169
Table 192 – Builtin put_unsigned_value2	170
Table 193 – Builtin ret_double_value	171
Table 194 – Builtin ret_double_value2	171
Table 195 – Builtin ret_float_value	172
Table 196 – Builtin ret_float_value2	172
Table 197 – Builtin ret_signed_value	173
Table 198 – Builtin ret_signed_value2	173
Table 199 – Builtin ret_unsigned_value	174
Table 200 – Builtin ret_unsigned_value2	174

Table 201 – Builtin vassign	175
Table 202 – Builtin discard_on_exit	176
Table 203 – Builtin save_on_exit	177
Table 204 – Builtin save_values	177
Table 205 – Builtin send_on_exit	178
Table 206 – Builtin abort	179
Table 207 – Builtin add_abort_method	179
Table 208 – Builtin add_abort_method	180
Table 209 – Builtin method_abort	181
Table 210 – Builtin pop_abort_method	181
Table 211 – Builtin process_abort	182
Table 212 – Builtin push_abort_method	182
Table 213 – Builtin remove_abort_method	183
Table 214 – Builtin remove_abort_method	183
Table 215 – Builtin remove_all_abort_methods	184
Table 216 – Builtin atof	184
Table 217 – Builtin atoi	185
Table 218 – Builtin BUILD_MESSAGE	185
Table 219 – Builtin dictionary_string	186
Table 220 – Builtin ftoa	186
Table 221 – Builtin get_comm_error_string	187
Table 222 – Builtin get_dictionary_string	187
Table 223 – Builtin get_enum_string	188
Table 224 – Builtin get_response_code_string	189
Table 225 – Builtin get_rspcode_string	190
Table 226 – Builtin get_rspcode_string_by_id	190
Table 227 – Builtin get_status_code_string	191
Table 228 – Builtin get_status_string	191
Table 229 – Builtin get_stddict_string	192
Table 230 – Builtin get_variable_string	193
Table 231 – Builtin itoa (version A)	193
Table 232 – Builtin itoa (version B)	194
Table 233 – Builtin strleft	194
Table 234 – Builtin strcmp	195
Table 235 – Builtin strlen	195
Table 236 – Builtin strlwr	196
Table 237 – Builtin strmid	197
Table 238 – Builtin strright	197
Table 239 – Builtin strstr	198
Table 240 – Builtin strtrim	198
Table 241 – Builtin strupr	199
Table 242 – Builtin trunc	199
Table 243 – Builtin AddTime	200

Table 244 – Builtin DATE_AND_TIME_VALUE_to_string	201
Table 245 – Builtin Date_to_DayOfMonth.....	201
Table 246 – Builtin DATE_to_days.....	202
Table 247 – Builtin Date_to_Month	202
Table 248 – Builtin DATE_to_string	203
Table 249 – Builtin Date_To_Time	203
Table 250 – Builtin Date_to_Year.....	203
Table 251 – Builtin days_to_DATE	204
Table 252 – Builtin DELAY_TIME.....	204
Table 253 – Builtin DiffTime.....	205
Table 254 – Builtin From_DATE_AND_TIME_VALUE.....	205
Table 255 – Builtin From_TIME_VALUE.....	206
Table 256 – Builtin GetCurrentDate	206
Table 257 – Builtin GetCurrentDateAndTime.....	206
Table 258 – Builtin GetCurrentTime	207
Table 259 – Builtin GET_TICK_COUNT	207
Table 260 – Builtin Make_Time	208
Table 261 – Builtin seconds_to_TIME_VALUE	208
Table 262 – Builtin seconds_to_TIME_VALUE	209
Table 263 – Builtin Time_To_Date	209
Table 264 – Builtin TIME_VALUE_to_Hour	210
Table 265 – Builtin TIME_VALUE_to_Minute.....	210
Table 266 – Builtin TIME_VALUE_to_Second	211
Table 267 – Builtin TIME_VALUE_to_seconds	211
Table 268 – Builtin TIME_VALUE_to_string	212
Table 269 – Builtin timet_to_string.....	213
Table 270 – Builtin timet_to_TIME_VALUE	213
Table 271 – Builtin timet_To_TIME_VALUE	214
Table 272 – Builtin timet_to_TIME_VALUE8	214
Table 273 – Builtin To_Date.....	214
Table 274 – Builtin To_Date_and_Time	215
Table 275 – Builtin To_Time	215
Table 276 – Builtin To_TIME_VALUE.....	216
Table 277 – Builtin To_TIME_VALUE8.....	216
Table 278 – Builtin get_comm_error.....	217
Table 279 – Builtin get_dds_error	218
Table 280 – Builtin get_response_code	219
Table 281 – Builtin abs	219
Table 282 – Builtin acos.....	220
Table 283 – Builtin asin	220
Table 284 – Builtin atan	221
Table 285 – Builtin ByteToDouble	221
Table 286 – Builtin ByteToFloat	222

Table 287 – Builtin ByteToLong	222
Table 288 – Builtin ByteToShort.....	223
Table 289 – Builtin cbrt	223
Table 290 – Builtin ceil	223
Table 291 – Builtin cos	224
Table 292 – Builtin cosh.....	224
Table 293 – Builtin DoubleToByte	225
Table 294 – Builtin drand	225
Table 295 – Builtin dseed	226
Table 296 – Builtin exp	226
Table 297 – Builtin FloatToByte	227
Table 298 – Builtin floor	227
Table 299 – Builtin fmod	228
Table 300 – Builtin fpclassify	228
Table 301 – Builtin is_NaN.....	229
Table 302 – Builtin log	229
Table 303 – Builtin log10	230
Table 304 – Builtin log2	230
Table 305 – Builtin LongToByte	231
Table 306 – Builtin nan	231
Table 307 – Builtin NaN_value.....	232
Table 308 – Builtin nanf	233
Table 309 – Builtin pow.....	233
Table 310 – Builtin round	233
Table 311 – Builtin ShortToByte.....	234
Table 312 – Builtin sin	234
Table 313 – Builtin sinh	235
Table 314 – Builtin sqrt.....	235
Table 315 – Builtin tan	236
Table 316 – Builtin tanh	236
Table 317 – Builtin get_date_lelem	237
Table 318 – Builtin get_date_lelem2	238
Table 319 – Builtin get_double_lelem	239
Table 320 – Builtin get_double_lelem2.....	240
Table 321 – Builtin get_float_lelem	240
Table 322 – Builtin get_float_lelem2	241
Table 323 – Builtin get_signed_lelem.....	242
Table 324 – Builtin get_signed_lelem2.....	243
Table 325 – Builtin get_string_lelem	244
Table 326 – Builtin get_string_lelem2	245
Table 327 – Builtin get_unsigned_lelem	246
Table 328 – Builtin get_unsigned_lelem2	247
Table 329 – Builtin ListDeleteElementAt	248

Table 330 – Builtin ListDeleteElementAt	248
Table 331 – Builtin ListDeleteElementAt2	249
Table 332 – Builtin ListInsert.....	250
Table 333 – Builtin ListInsert.....	250
Table 334 – Builtin ListInsert2.....	251
Table 335 – Builtin DICT_ID	252
Table 336 – Builtin get_block_instance_by_object_index	252
Table 337 – Builtin get_block_instance_by_tag	253
Table 338 – Builtin get_block_instance_count.....	254
Table 339 – Builtin get_resolve_status.....	255
Table 340 – Builtin ITEM_ID	255
Table 341 – Builtin MEMBER_ID.....	256
Table 342 – Builtin ObjectReference	256
Table 343 – Builtin resolve_array_ref.....	257
Table 344 – Builtin resolve_array_ref2.....	258
Table 345 – Builtin resolve_block_ref.....	259
Table 346 – Builtin resolve_block_ref2.....	260
Table 347 – Builtin resolve_list_ref	260
Table 348 – Builtin resolve_local_ref	261
Table 349 – Builtin resolve_local_ref2.....	261
Table 350 – Builtin resolve_param_list_ref.....	262
Table 351 – Builtin resolve_param_ref	263
Table 352 – Builtin resolve_param_ref2.....	263
Table 353 – Builtin resolve_record_ref.....	264
Table 354 – Builtin resolve_record_ref2.....	265
Table 355 – Builtin VARID	265
Table 356 – Builtin abortTransferPort.....	266
Table 357 – Builtin closeTransferPort	266
Table 358 – Builtin fGetByte	267
Table 359 – Builtin get_transfer_status	267
Table 360 – Builtin openTransferPort.....	268
Table 361 – Builtin readItemFromDevice.....	268
Table 362 – Builtin writeItemToDevice	269
Table 363 – Builtin browselIdentity	269
Table 364 – Builtin re_read_file	270
Table 365 – Builtin re_write_file	270
Table 366 – Builtin GET_DD_REVISION.....	271
Table 367 – Builtin GET_DEVICE_REVISION.....	271
Table 368 – Builtin GET_DEVICE_TYPE	272
Table 369 – Builtin GET_MANUFACTURER.....	272
Table 370 – Builtin _ERROR.....	273
Table 371 – Builtin _TRACE	273
Table 372 – Builtin _WARNING.....	274

Table 373 – Builtin LOG_MESSAGE	274
Table 374 – Referencable EDD item attributes	276
Table 375 – Contents of the return codes description table	279
Table 376 – Return code descriptions	279
Table 377 – Return code descriptions	279

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

INTERNATIONAL ELECTROTECHNICAL COMMISSION

**DEVICES AND INTEGRATION IN ENTERPRISE SYSTEMS –
FUNCTION BLOCKS (FB) FOR PROCESS CONTROL AND
ELECTRONIC DEVICE DESCRIPTION LANGUAGE (EDDL) –****Part 5: EDDL Builtin library****FOREWORD**

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
- 8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61804-5 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2015. This edition constitutes a technical revision.

This edition was developed by merging material from multiple variants of existing EDDL specifications including those from FieldComm Group (Foundation™ Fieldbus¹, HART®²), PROFIBUS™³ Nutzerorganisation e.V. (PNO), and ISA100_Wireless™⁴ Compliance Institute (ISA100 WCI). As a result, the formatting and numbering of this edition may be different from any of the individual specifications from which this edition was derived.

This edition includes the following significant technical changes with respect to the previous edition:

- Communication profiles ISA100 and GPE were added.
- The following builtins have been deprecated:
 - ABORT_ON_NO_DEVICE
 - IGNORE_NO_DEVICE
 - RETRY_ON_NO_DEVICE
 - XMTR_ABORT_ON_NO_DEVICE
 - XMTR_IGNORE_NO_DEVICE
 - XMTR_RETRY_ON_NO_DEVICE
 - get_status_code_string

The text of this International Standard is based on the following documents:

CDV	Report on voting
65E/634/CDV	65E/691/RVC

Full information on the voting for the approval of this International Standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts in the IEC 61804 series, published under the general title *Devices and integration in enterprise systems – Function blocks (FB) for process control and electronic device description language (EDDL)*, can be found on the IEC website.

Future standards in this series will carry the new general title as cited above. Titles of existing standards in this series will be updated at the time of the next edition.

¹ FOUNDATION™ Fieldbus is the trademark of FieldComm Group. This information is given for the convenience of users of this document and does not constitute an endorsement by IEC of the product named. Equivalent products may be used if they can be shown to lead to the same results.

² HART® is the registered trademark of FieldComm Group. This information is given for the convenience of users of this document and does not constitute an endorsement by IEC of the product named. Equivalent products may be used if they can be shown to lead to the same results.

³ PROFIBUS and PROFINET are the trademarks of the PROFIBUS Nutzerorganisation e.V. This information is given for the convenience of users of this document and does not constitute an endorsement by IEC of the product named. Equivalent products may be used if they can be shown to lead to the same results.

⁴ ISA100_Wireless™ is the trademark of ISA100 Wireless Compliance Institute. This information is given for the convenience of users of this document and does not constitute an endorsement by IEC of the product named. Equivalent products may be used if they can be shown to lead to the same results.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

INTRODUCTION

This part of IEC 61804

- contains a list of builtins that can be used inside of EDDL methods;
- defines the functionality and parameters for each builtin;
- describes how EDDL variables can be referenced from inside EDDL methods;

EDDL features are limited by profile for each of the communication technologies. The descriptions in this part of IEC 61804 refer to these features in a general sense and not all communication technologies will support all of the features described. The profile definitions in the overview of this part and in IEC 61804-3 are referred to in order to understand the features supported by each communication technology.

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

DEVICES AND INTEGRATION IN ENTERPRISE SYSTEMS – FUNCTION BLOCKS (FB) FOR PROCESS CONTROL AND ELECTRONIC DEVICE DESCRIPTION LANGUAGE (EDDL) –

Part 5: EDDL Builtin library

1 Scope

This part of IEC 61804 specifies the EDDL builtin library and provides the profiles of the various fieldbuses.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61804-3:2020, *Devices and integration in enterprise systems – Function blocks (FB) for process control and electronic device description language (EDDL) – Part 3: EDDL syntax and semantics*

ISO/IEC 9899, *Information technology – Programming Languages – C*

IEEE 754, *IEEE Standard for Binary Floating-Point Arithmetic*

3 Terms, definitions, acronyms and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 61804-3 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at <http://www.electropedia.org/>
- ISO Online browsing platform: available at <http://www.iso.org/obp>

3.1.1

method-variable

variable with scope within a method

3.2 Acronyms and abbreviated terms

CP	Communication Profile
CPF	Communication Profile Family
CS	Communication Server
DD	Device Description
EDD	Electronic Device Description
EDDL	Electronic Device Description Language

FF	Foundation Fieldbus
GPE	Generic Protocol Extension
HART	HART® Communication Foundation
I/O	Input/output
ID	Identifier
ISA100	ISA100_Wireless™
PB	PROFIBUS
PI	PROFIBUS & PROFINET ³ International
PN	PROFINET
PNO	PROFIBUS Nutzerorganisation

4 Conventions for builtin descriptions

The following conventions apply for the builtin overview.

- Table 1 defines the format for the builtin overview table.
- Table 2 defines the contents of the columns of Table 1.
- EDDL basic constructs and attributes referenced in this document are written in upper case letters.

Table 1 – Format for the builtin overview table

Reference	Builtin	Communication profiles

Table 2 – Contents of the builtin overview table

Column	Text	Meaning
Reference		Reference to the respective subclause that specifies the builtin.
Builtin		Name of the builtin.
Communication profiles	✓	The builtin is supported by the respective Communication profile.
	x	The builtin is not supported by the respective Communication profile.
	d	The usage of the builtin is deprecated.

The following conventions apply for the builtin definitions.

- Table 3 defines the format for the builtins lexical element tables.
- Table 4 defines the contents of the columns of Table 3.
- EDDL basic constructs and attributes referenced in this document are written in upper case letters.

Table 3 – Format for the builtins lexical element tables

Parameter name	Type	Direction	Usage	Description

Table 4 – Contents of the lexical element table

Column	Text	Meaning
Parameter name	<name>	The name of the formal parameter.
	"<return>"	The return value of the builtin.
Type	data type	Abstract data type (basic or derived) of the formal parameter.
Direction	I	An input parameter.
	O	An output parameter is implicitly passed by reference so that the called builtin function can modify the value. The return value of a builtin is passed by value, if any.
	I/O	An input output parameter is implicitly passed by reference so that the called builtin function can modify the value.
	—	There is no returned value (VOID).
Usage	m	Formal parameter is mandatory.
	o	Formal parameter is optional.
	c	Formal parameter is conditional.
	—	There is no returned value (VOID).
Description	<text>	Descriptive text for the parameter.
	—	There is no returned value (VOID).

5 Builtin categories

5.1 Overview

Table 5 provides an alphabetical overview of the EDDL builtin library and specify the profile per communication technology.

Table 5 – Builtin overview

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.17.1	ERROR	✓	✓	✓	✓	✓	✓	✓
6.17.2	_TRACE	✓	✓	✓	✓	✓	✓	✓
6.17.3	_WARNING	✓	✓	✓	✓	✓	✓	✓
6.7.1	abort	✓	✗	✓	✓	✗	✓	✓
6.3.1	abort_on_all_comm_errors	✗	✓	✗	✗	✓	✗	✗
6.3.2	ABORT_ON_ALL_COMM_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.3	ABORT_ON_ALL_DEVICE_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.4	ABORT_ON_ALL_RESPONSE_CODES	✗	✗	✗	✓	✗	✗	✗
6.3.5	abort_on_all_response_codes	✗	✓	✗	✗	✓	✗	✗
6.3.6	ABORT_ON_COMM_ERROR	✗	✗	✗	✓	✗	✗	✗
6.3.7	abort_on_comm_error	✗	✓	✗	✗	✓	✗	✗
6.3.8	ABORT_ON_COMM_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.9	ABORT_ON_DEVICE_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.10	ABORT_ON_NO_DEVICE (deprecated)	✗	✗	✗	d	✗	✗	✗

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.11	ABORT_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.12	abort_on_response_code	x	✓	x	x	✓	x	x
6.14.1	abortTransferPort	x	x	x	✓	x	x	x
6.11.1	abs	✓	✓	✓	✓	✓	✓	✓
6.2.1	ACKNOWLEDGE	✓	x	✓	✓	x	✓	✓
6.2.2	acknowledge	✓	x	✓	✓	x	✓	✓
6.11.2	acos	✓	✓	✓	✓	✓	✓	✓
6.7.2	add_abort_method (version A)	✓	x	✓	✓	✓	✓	✓
6.7.3	add_abort_method (version B)	x	✓	x	x	✓	x	x
6.9.1	AddTime	✓	✓	✓	✓	✓	✓	✓
6.11.3	asin	✓	✓	✓	✓	✓	✓	✓
6.5.1	assign	x	✓	x	x	✓	x	x
6.5.2	assign_double	✓	x	✓	✓	x	✓	✓
6.5.3	assign_float	✓	x	✓	✓	x	✓	✓
6.5.4	assign_int	✓	x	✓	✓	x	✓	✓
6.5.5	assign_var	✓	x	✓	✓	x	✓	✓
6.5.6	assign2	x	✓	x	x	✓	x	x
6.11.4	atan	✓	✓	✓	✓	✓	✓	✓
6.8.1	atof	✓	x	✓	✓	x	✓	✓
6.8.2	atoi	✓	x	✓	✓	x	✓	✓
6.15.1	browseIdentity	x	x	x	✓	x	x	x
6.8.3	BUILD_MESSAGE	✓	✓	✓	✓	✓	✓	✓
6.11.5	ByteToDouble	✓	x	✓	x	x	✓	✓
6.11.6	ByteToFloat	✓	x	✓	x	x	✓	✓
6.11.7	ByteToLong	✓	x	✓	x	x	✓	✓
6.11.8	ByteToShort	✓	x	✓	x	x	✓	✓
6.11.9	cbrt	✓	✓	✓	✓	✓	✓	✓
6.11.10	ceil	✓	✓	✓	✓	✓	✓	✓
6.14.2	closeTransferPort	x	x	x	✓	x	x	x
6.11.11	cos	✓	✓	✓	✓	✓	✓	✓
6.11.12	cosh	✓	✓	✓	✓	✓	✓	✓
6.5.7	dassign	✓	x	✓	✓	x	✓	✓
6.9.2	DATE_AND_TIME_VALUE_to_string	✓	x	✓	✓	x	✓	✓
6.9.3	Date_to_DayOfMonth	✓	x	✓	✓	x	✓	✓
6.9.4	DATE_to_days	✓	x	✓	✓	x	✓	✓
6.9.5	Date_to_Month	✓	x	✓	✓	x	✓	✓
6.9.6	DATE_to_string	✓	x	✓	✓	x	✓	✓
6.9.7	Date_To_Time	✓	x	✓	✓	x	✓	✓
6.9.8	Date_to_Year	✓	x	✓	✓	x	✓	✓
6.9.9	days_to_DATE	✓	x	✓	✓	x	✓	✓

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.2.3	DELAY	✓	✗	✓	✓	✗	✓	✓
6.2.4	delay	✓	✗	✓	✓	✗	✓	✓
6.9.10	DELAY_TIME	✓	✗	✓	✓	✗	✓	✓
6.2.5	delayfor	✗	✓	✗	✗	✓	✗	✗
6.2.6	delayfor2	✗	✓	✗	✗	✓	✗	✗
6.13.1	DICT_ID	✗	✓	✗	✗	✓	✗	✗
6.8.4	dictionary_string	✓	✗	✓	✓	✗	✓	✓
6.9.11	DiffTime	✓	✓	✓	✓	✓	✓	✓
6.6.1	discard_on_exit	✓	✓	✓	✓	✓	✓	✓
6.2.7	DISPLAY	✓	✗	✓	✓	✗	✓	✓
6.2.8	display	✓	✗	✓	✓	✗	✓	✓
6.2.9	display_bitenum	✓	✗	✓	✗	✗	✓	✓
6.2.10	display_builtin_error	✗	✓	✗	✗	✓	✗	✗
6.2.11	display_comm_error	✗	✓	✗	✗	✓	✗	✗
6.2.12	display_comm_status	✗	✗	✗	✓	✗	✗	✗
6.2.13	display_device_status	✗	✗	✗	✓	✗	✗	✗
6.2.14	display_dynamics	✗	✓	✗	✗	✓	✗	✗
6.2.15	display_dynamics2	✗	✓	✗	✗	✓	✗	✗
6.2.16	display_message	✗	✓	✗	✗	✓	✗	✗
6.2.17	display_message2	✗	✓	✗	✗	✓	✗	✗
6.2.18	display_response_code	✗	✓	✗	✗	✓	✗	✗
6.2.19	display_response_status	✗	✗	✗	✓	✗	✗	✗
6.2.20	display_xmtr_status	✗	✗	✗	✓	✗	✗	✗
6.11.13	DoubleToByte	✓	✗	✓	✗	✗	✓	✓
6.11.14	drand	✓	✓	✓	✓	✓	✓	✓
6.11.15	dseed	✓	✓	✓	✓	✓	✓	✓
6.2.21	edit_device_value	✗	✓	✗	✗	✓	✗	✗
6.2.22	edit_device_value2	✗	✓	✗	✗	✓	✗	✗
6.2.23	edit_local_value	✗	✓	✗	✗	✓	✗	✗
6.2.24	edit_local_value2	✗	✓	✗	✗	✓	✗	✗
6.11.16	exp	✓	✓	✓	✓	✓	✓	✓
6.3.13	ext_send_command	✗	✗	✗	✓	✗	✗	✗
6.3.14	ext_send_command_trans	✗	✗	✗	✓	✗	✗	✗
6.3.15	fail_on_all_comm_errors	✗	✓	✗	✗	✓	✗	✗
6.3.16	fail_on_all_response_codes	✗	✓	✗	✗	✓	✗	✗
6.3.17	fail_on_comm_error	✗	✓	✗	✗	✓	✗	✗
6.3.18	fail_on_response_code	✗	✓	✗	✗	✓	✗	✗
6.5.8	fassign	✓	✗	✓	✓	✗	✓	✓
6.14.2	fGetByte	✗	✗	✗	✓	✗	✗	✗
6.4.1	fgetval	✓	✗	✓	✓	✗	✓	✓

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.5.9	float_value	✓	✗	✓	✓	✗	✓	✓
6.11.17	FloatToByte	✓	✗	✓	✗	✗	✓	✓
6.11.18	floor	✓	✓	✓	✓	✓	✓	✓
6.11.19	fmod	✓	✓	✓	✓	✓	✓	✓
6.11.20	fpclassify	✓	✓	✓	✓	✓	✓	✓
6.9.12	From_DATE_AND_TIME_VALUE	✓	✗	✓	✓	✗	✓	✓
6.9.13	From_TIME_VALUE	✓	✗	✓	✓	✗	✓	✓
6.4.2	fsetval	✓	✗	✓	✓	✓	✓	✓
6.8.5	ftoa	✓	✗	✓	✗	✗	✓	✓
6.5.10	fvar_value	✓	✗	✓	✓	✗	✓	✓
6.2.25	get_acknowledgement	✗	✓	✗	✗	✓	✗	✗
6.2.26	get_acknowledgement2	✗	✓	✗	✗	✓	✗	✗
6.13.2	get_block_instance_by_object_index	✗	✓	✗	✗	✓	✗	✗
6.13.3	get_block_instance_by_tag	✗	✓	✗	✗	✓	✗	✗
6.13.4	get_block_instance_count	✗	✓	✗	✗	✓	✗	✗
6.10.1	get_comm_error	✗	✓	✗	✗	✓	✗	✗
6.8.6	get_comm_error_string	✗	✓	✗	✗	✓	✗	✗
6.4.3	get_date	✗	✓	✗	✗	✓	✗	✗
6.12.1	get_date_lelem	✗	✓	✗	✗	✓	✗	✗
6.12.2	get_date_lelem2	✗	✓	✗	✗	✓	✗	✗
6.5.11	get_date_value	✗	✓	✗	✗	✓	✗	✗
6.5.12	get_date_value2	✗	✓	✗	✗	✓	✗	✗
6.16.1	GET_DD_REVISION	✓	✗	✓	✗	✗	✓	✓
6.10.2	get_dds_error	✗	✓	✗	✗	✓	✗	✗
6.2.27	GET_DEV_VAR_VALUE	✓	✗	✓	✓	✗	✓	✓
6.2.28	get_dev_var_value	✓	✗	✓	✓	✗	✓	✓
6.16.2	GET_DEVICE_REVISION	✓	✗	✓	✗	✗	✓	✓
6.16.3	GET_DEVICE_TYPE	✓	✗	✓	✗	✗	✓	✓
6.8.7	get_dictionary_string	✓	✗	✓	✓	✗	✓	✓
6.4.4	get_double	✗	✓	✗	✗	✓	✗	✗
6.12.3	get_double_lelem	✗	✓	✗	✗	✓	✗	✗
6.12.4	get_double_lelem2	✗	✓	✗	✗	✓	✗	✗
6.5.13	get_double_value	✗	✓	✗	✗	✓	✗	✗
6.5.14	get_double_value2	✗	✓	✗	✗	✓	✗	✗
6.8.8	get_enum_string	✓	✗	✓	✓	✗	✓	✓
6.4.5	get_float	✗	✓	✗	✗	✓	✗	✗
6.12.5	get_float_lelem	✗	✓	✗	✗	✓	✗	✗
6.12.6	get_float_lelem2	✗	✓	✗	✗	✓	✗	✗
6.5.15	get_float_value	✗	✓	✗	✗	✓	✗	✗
6.5.16	get_float_value2	✗	✓	✗	✗	✓	✗	✗

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.2.29	GET_LOCAL_VAR_VALUE	✓	x	✓	✓	x	✓	✓
6.2.30	get_local_var_value	✓	x	✓	✓	x	✓	✓
6.16.4	GET_MANUFACTURER	✓	x	✓	✓	x	✓	✓
6.3.19	get_more_status	x	x	x	✓	x	x	x
6.13.5	get_resolve_status	x	✓	x	x	✓	x	x
6.10.3	get_response_code	x	✓	x	x	✓	x	x
6.8.9	get_response_code_string	x	✓	x	x	✓	x	x
6.8.10	get_rspcode_string	x	x	x	✓	x	x	x
6.8.11	get_rspcode_string_by_id	✓	x	✓	x	x	✓	✓
6.4.6	get_signed	x	✓	x	x	✓	x	x
6.12.7	get_signed_lelem	x	✓	x	x	✓	x	x
6.12.8	get_signed_lelem2	x	✓	x	x	✓	x	x
6.5.17	get_signed_value	x	✓	x	x	✓	x	x
6.5.18	get_signed_value2	x	✓	x	x	✓	x	x
6.8.12	get_status_code_string(deprecated)	x	x	x	d	x	x	x
6.8.13	get_status_string	x	✓	x	x	✓	x	x
6.8.14	get_stddict_string	x	✓	x	x	✓	x	x
6.4.7	get_string	x	✓	x	x	✓	x	x
6.12.9	get_string_lelem	x	✓	x	x	✓	x	x
6.12.10	get_string_lelem2	x	✓	x	x	✓	x	x
6.5.19	get_string_value	x	✓	x	x	✓	x	x
6.5.20	get_string_value2	x	✓	x	x	✓	x	x
6.9.17	GET_TICK_COUNT	✓	x	✓	x	x	✓	✓
6.14.4	get_transfer_status	x	x	x	✓	x	x	x
6.4.8	get_unsigned	x	✓	x	x	✓	x	x
6.12.11	get_unsigned_lelem	x	✓	x	x	✓	x	x
6.12.12	get_unsigned_lelem2	x	✓	x	x	✓	x	x
6.5.21	get_unsigned_value	x	✓	x	x	✓	x	x
6.5.22	get_unsigned_value2	x	✓	x	x	✓	x	x
6.8.15	get_variable_string	✓	x	✓	x	x	✓	✓
6.9.14	GetCurrentDate	✓	✓	✓	x	✓	✓	✓
6.9.15	GetCurrentDateAndTime	✓	✓	✓	x	✓	✓	✓
6.9.16	GetCurrentTime	✓	✓	✓	✓	✓	✓	✓
6.5.23	iassign	✓	x	✓	✓	x	✓	✓
6.4.9	igetval	✓	x	✓	✓	x	✓	✓
6.3.20	IGNORE_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.21	IGNORE_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.22	IGNORE_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.23	IGNORE_COMM_ERROR	x	x	x	✓	x	x	x
6.3.24	IGNORE_COMM_STATUS	x	x	x	✓	x	x	x

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.25	IGNORE_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.26	IGNORE_NO_DEVICE (deprecated)	x	x	x	d	x	x	x
6.3.27	IGNORE_RESPONSE_CODE	x	x	x	✓	x	x	x
6.5.24	int_value	✓	x	✓	✓	x	✓	✓
6.11.21	is_NaN	✓	✓	✓	x	✓	✓	✓
6.4.10	isetval	✓	x	✓	✓	x	✓	✓
6.4.11	isOffline	✓	✓	✓	✓	✓	✓	✓
6.13.6	ITEM_ID	x	✓	x	x	x	x	x
6.8.16	itoa (version A)	✓	x	✓	x	x	✓	✓
6.8.17	itoa (version B)	x	x	x	x	x	x	x
6.5.25	ivar_value	✓	x	✓	✓	x	✓	✓
6.5.26	lassign	✓	x	✓	x	x	✓	✓
6.4.12	lgetval	✓	x	✓	x	x	✓	✓
6.12.13	ListDeleteElementAt (version A)	✓	x	✓	✓	x	✓	✓
6.12.14	ListDeleteElementAt (version B)	x	✓	x	x	✓	x	x
6.12.15	ListDeleteElementAt2	x	✓	x	x	✓	x	x
6.12.16	ListInsert (version A)	✓	x	✓	✓	x	✓	✓
6.12.17	ListInsert (version B)	x	✓	x	x	✓	x	x
6.12.18	ListInsert2	x	✓	x	x	✓	x	x
6.11.22	log	✓	✓	✓	✓	✓	✓	✓
6.17.4	LOG_MESSAGE	✓	x	✓	✓	x	✓	✓
6.11.23	log10	✓	✓	✓	✓	✓	✓	✓
6.11.24	log2	✓	✓	✓	✓	✓	✓	✓
6.5.27	long_value	✓	x	✓	x	x	✓	✓
6.11.25	LongToByte	✓	x	✓	x	x	✓	✓
6.4.13	lsetval	✓	x	✓	x	x	✓	✓
6.5.28	lvar_value	✓	x	✓	x	x	✓	✓
6.9.18	Make_Time	✓	✓	✓	✓	✓	✓	✓
6.13.7	MEMBER_ID	x	✓	x	x	✓	x	x
6.2.31	Menu	✓	x	✓	✓	x	✓	✓
6.2.32	MenuDisplay (version A)	✓	x	✓	✓	x	✓	✓
6.2.33	MenuDisplay (version B)	x	✓	x	x	✓	x	x
6.7.4	method_abort	x	✓	x	x	✓	x	x
6.11.26	nan	✓	✓	✓	✓	✓	✓	✓
6.11.27	NaN_value	✓	✓	✓	x	✓	✓	✓
6.11.28	nanf	✓	✓	✓	✓	✓	✓	✓
6.13.8	ObjectReference	✓	x	✓	x	x	✓	✓
6.14.5	openTransferPort	x	x	x	✓	x	x	x
6.7.5	pop_abort_method	✓	x	✓	✓	x	✓	✓
6.11.29	pow	✓	✓	✓	✓	✓	✓	✓

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.7.6	process_abort	✓	✗	✓	✓	✗	✓	✓
6.7.7	push_abort_method	✓	✗	✓	✓	✗	✓	✓
6.4.14	put_date	✗	✓	✗	✗	✓	✗	✗
6.5.29	put_date_value	✗	✓	✗	✗	✓	✗	✗
6.5.30	put_date_value2	✗	✓	✗	✗	✓	✗	✗
6.4.15	put_double	✗	✓	✗	✗	✓	✗	✗
6.5.31	put_double_value	✗	✓	✗	✗	✓	✗	✗
6.5.32	put_double_value2	✗	✓	✗	✗	✓	✗	✗
6.4.16	put_float	✗	✓	✗	✗	✓	✗	✗
6.5.33	put_float_value	✗	✓	✗	✗	✓	✗	✗
6.5.34	put_float_value2	✗	✓	✗	✗	✓	✗	✗
6.2.34	PUT_MESSAGE	✓	✗	✓	✓	✗	✓	✓
6.2.35	put_message	✓	✗	✓	✓	✗	✓	✓
6.4.17	put_signed	✗	✓	✗	✗	✓	✗	✗
6.5.35	put_signed_value	✗	✓	✗	✗	✓	✗	✗
6.5.36	put_signed_value2	✗	✓	✗	✗	✓	✗	✗
6.4.18	put_string	✗	✓	✗	✗	✓	✗	✗
6.5.37	put_string_value	✗	✓	✗	✗	✓	✗	✗
6.5.38	put_string_value2	✗	✓	✗	✗	✓	✗	✗
6.4.19	put_unsigned	✗	✓	✗	✗	✓	✗	✗
6.5.39	put_unsigned_value	✗	✓	✗	✗	✓	✗	✗
6.5.40	put_unsigned_value2	✗	✓	✗	✗	✓	✗	✗
6.15.2	re_read_file	✗	✗	✗	✓	✗	✗	✗
6.15.3	re_write_file	✗	✗	✗	✓	✗	✗	✗
6.3.28	read_value	✗	✓	✗	✗	✓	✗	✗
6.3.29	read_value2	✗	✓	✗	✗	✓	✗	✗
6.3.30	ReadCommand	✓	✗	✓	✗	✗	✓	✓
6.14.6	readItemFromDevice	✗	✗	✗	✓	✗	✗	✗
6.7.8	remove_abort_method (version A)	✓	✗	✓	✓	✗	✓	✓
6.7.9	remove_abort_method (version B)	✗	✓	✗	✗	✓	✗	✗
6.7.10	remove_all_abort_methods	✓	✓	✓	✓	✓	✓	✓
6.13.9	resolve_array_ref	✗	✓	✗	✗	✓	✗	✗
6.13.10	resolve_array_ref2	✗	✓	✗	✗	✓	✗	✗
6.13.11	resolve_block_ref	✗	✓	✗	✗	✓	✗	✗
6.13.12	resolve_block_ref2	✗	✓	✗	✗	✓	✗	✗
6.13.13	resolve_list_ref	✗	✓	✗	✗	✓	✗	✗
6.13.14	resolve_local_ref	✗	✓	✗	✗	✓	✗	✗
6.13.15	resolve_local_ref2	✗	✓	✗	✗	✓	✗	✗
6.13.16	resolve_param_list_ref	✗	✓	✗	✗	✓	✗	✗
6.13.17	resolve_param_ref	✗	✓	✗	✗	✓	✗	✗

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.13.18	resolve_param_ref2	x	✓	x	x	✓	x	x
6.13.19	resolve_record_ref	x	✓	x	x	✓	x	x
6.13.20	resolve_record_ref2	x	✓	x	x	✓	x	x
6.5.41	ret_double_value	x	✓	x	x	✓	x	x
6.5.42	ret_double_value2	x	✓	x	x	✓	x	x
6.5.43	ret_float_value	x	✓	x	x	✓	x	x
6.5.44	ret_float_value2	x	✓	x	x	✓	x	x
6.5.45	ret_signed_value	x	✓	x	x	✓	x	x
6.5.46	ret_signed_value2	x	✓	x	x	✓	x	x
6.5.47	ret_unsigned_value	x	✓	x	x	✓	x	x
6.5.48	ret_unsigned_value2	x	✓	x	x	✓	x	x
6.3.31	retry_on_all_comm_errors	x	✓	x	x	✓	x	x
6.3.32	RETRY_ON_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.33	RETRY_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.34	RETRY_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.35	retry_on_all_response_codes	x	✓	x	x	✓	x	x
6.3.36	RETRY_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.37	retry_on_comm_error	x	✓	x	x	✓	x	x
6.3.38	RETRY_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.39	RETRY_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.40	RETRY_ON_NO_DEVICE (deprecated)	d	x	d	d	x	d	d
6.3.41	RETRY_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.42	retry_on_response_code	x	✓	x	x	✓	x	x
6.11.30	round	✓	✓	✓	✓	✓	✓	✓
6.6.2	save_on_exit	✓	✓	✓	x	✓	✓	✓
6.6.3	save_values	✓	x	✓	✓	x	✓	✓
6.9.19	seconds_to_TIME_VALUE	✓	x	✓	✓	x	✓	✓
6.9.20	seconds_to_TIME_VALUE8	✓	✓	✓	x	✓	✓	✓
6.2.36	SELECT_FROM_LIST	✓	x	✓	✓	x	✓	✓
6.2.37	select_from_list	✓	x	✓	✓	x	✓	✓
6.2.38	select_from_menu	x	✓	x	x	✓	x	x
6.2.39	select_from_menu2	x	✓	x	x	✓	x	x
6.3.43	send	x	x	x	✓	x	x	x
6.3.44	send_all_values	x	✓	x	x	✓	x	x
6.3.45	send_command	x	x	x	✓	x	x	x
6.3.46	send_command_trans	x	x	x	✓	x	x	x
6.6.4	send_on_exit	✓	✓	✓	x	✓	✓	✓
6.3.47	send_trans	x	x	x	✓	x	x	x
6.3.48	send_value	x	✓	x	x	✓	x	x
6.3.49	send_value2	x	✓	x	x	✓	x	x

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.50	SET_NUMBER_OF_RETRIES	✓	✗	✓	✓	✗	✓	✓
6.4.20	sgetval	✓	✗	✓	✓	✗	✓	✓
6.11.31	ShortToByte	✓	✗	✓	✗	✗	✓	✓
6.11.32	sin	✓	✓	✓	✓	✓	✓	✓
6.11.33	sinh	✓	✓	✓	✓	✓	✓	✓
6.11.34	sqrt	✓	✓	✓	✓	✓	✓	✓
6.4.21	ssetval	✓	✗	✓	✓	✗	✓	✓
6.8.19	strcmp	✓	✓	✓	✓	✓	✓	✓
6.8.18	strleft	✓	✓	✓	✗	✓	✓	✓
6.8.20	strlen	✓	✓	✓	✓	✓	✓	✓
6.8.21	strlwr	✓	✓	✓	✓	✓	✓	✓
6.8.22	strmid	✓	✓	✓	✓	✓	✓	✓
6.8.23	strright	✓	✓	✓	✗	✓	✓	✓
6.8.24	strstr	✓	✓	✓	✓	✓	✓	✓
6.8.25	strtrim	✓	✓	✓	✓	✓	✓	✓
6.8.26	strupr	✓	✓	✓	✓	✓	✓	✓
6.11.35	tan	✓	✓	✓	✓	✓	✓	✓
6.11.36	tanh	✓	✓	✓	✓	✓	✓	✓
6.9.21	Time_To_Date	✓	✗	✓	✓	✗	✓	✓
6.9.22	TIME_VALUE_to_Hour	✓	✓	✓	✓	✓	✓	✓
6.9.23	TIME_VALUE_to_Minute	✓	✓	✓	✓	✓	✓	✓
6.9.24	TIME_VALUE_to_Second	✓	✓	✓	✓	✓	✓	✓
6.9.25	TIME_VALUE_to_seconds	✓	✓	✓	✓	✓	✓	✓
6.9.26	TIME_VALUE_to_string	✓	✓	✓	✓	✓	✓	✓
6.9.27	timet_to_string	✓	✓	✓	✓	✓	✓	✓
6.9.28	timet_to_TIME_VALUE	✓	✗	✓	✗	✗	✓	✓
6.9.29	timet_To_TIME_VALUE	✗	✗	✗	✓	✗	✗	✗
6.9.30	timet_to_TIME_VALUE8	✓	✓	✓	✗	✓	✓	✓
6.9.31	To_Date	✓	✗	✓	✓	✗	✓	✓
6.9.32	To_Date_and_Time	✓	✗	✓	✗	✗	✓	✓
6.9.33	To_Time	✓	✗	✓	✓	✗	✓	✓
6.9.34	To_TIME_VALUE	✓	✗	✓	✓	✗	✓	✓
6.9.35	To_TIME_VALUE8	✓	✓	✓	✗	✓	✓	✓
6.8.27	trunc	✓	✓	✓	✓	✓	✓	✓
6.13.21	VARID	✓	✗	✓	✓	✗	✓	✓
6.5.49	vassign	✓	✗	✓	✓	✗	✓	✓
6.3.51	WriteCommand	✓	✗	✓	✗	✗	✓	✓
6.14.7	writeltemToDevice	✗	✗	✗	✓	✗	✗	✗
6.3.52	XMTR_ABORT_ON_ALL_COMM_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.53	XMTR_ABORT_ON_ALL_DATA	✗	✗	✗	✓	✗	✗	✗

Reference	Builtin	Communication profiles						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.54	XMTR_ABORT_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.55	XMTR_ABORT_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.56	XMTR_ABORT_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.57	XMTR_ABORT_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.58	XMTR_ABORT_ON_DATA	x	x	x	✓	x	x	x
6.3.59	XMTR_ABORT_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.60	XMTR_ABORT_ON_NO_DEVICE (deprecated)	x	x	x	d	x	x	x
6.3.61	XMTR_ABORT_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.62	XMTR_IGNORE_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.63	XMTR_IGNORE_ALL_DATA	x	x	x	✓	x	x	x
6.3.64	XMTR_IGNORE_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.65	XMTR_IGNORE_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.66	XMTR_IGNORE_COMM_ERROR	x	x	x	✓	x	x	x
6.3.67	XMTR_IGNORE_COMM_STATUS	x	x	x	✓	x	x	x
6.3.68	XMTR_IGNORE_DATA	x	x	x	✓	x	x	x
6.3.69	XMTR_IGNORE_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.70	XMTR_IGNORE_NO_DEVICE (deprecated)	x	x	x	d	x	x	x
6.3.71	XMTR_IGNORE_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.72	XMTR_RETRY_ON_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.73	XMTR_RETRY_ON_ALL_DATA	x	x	x	✓	x	x	x
6.3.74	XMTR_RETRY_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.75	XMTR_RETRY_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.76	XMTR_RETRY_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.77	XMTR_RETRY_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.78	XMTR_RETRY_ON_DATA	x	x	x	✓	x	x	x
6.3.79	XMTR_RETRY_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.80	XMTR_RETRY_ON_NO_DEVICE (deprecated)	x	x	x	d	x	x	x
6.3.81	XMTR_RETRY_ON_RESPONSE_CODE	x	x	x	✓	x	x	x

There are the following special categories of builtins:

- User interface builtins;
- Communication builtins;
- Action builtins.

Table 6 shows restrictions to the usage of builtins, depending on their category.

Table 6 – Usage of builtins

Builtin usage	User interface builtins	Communication builtins	Action builtins
Methods referenced in user interface menus	Allowed	Allowed	Not allowed
PRE_EDIT_ACTIONS	Allowed	Not recommended	Allowed ^a
POST_EDIT_ACTIONS	Allowed	Not recommended	Allowed ^a
REFRESH_ACTIONS	Not recommended	GRAPH and CHART: allowed Others: not recommended	Allowed ^a
PRE_READ_ACTIONS	Not recommended	Allowed	Allowed ^a
POST_READ_ACTIONS	Not recommended	Allowed	Allowed ^a
PRE_WRITE_ACTIONS	Not recommended	Allowed	Allowed ^a
POST_WRITE_ACTIONS	Not recommended	Allowed	Allowed ^a
INIT_ACTIONS	Not recommended	Allowed	Not allowed
EXIT_ACTIONS	Not recommended	Allowed	Not allowed
^a Shall only be used for actions called from a VARIABLE.			

5.2 User interface builtins

User interface builtins display information and in addition some of them require interaction from the user. The list of user interface builtins are listed in Table 7.

Table 7 – User interface builtins

Builtin name
ACKNOWLEDGE
acknowledge
DELAY
delay
delayfor
delayfor2
DISPLAY
display
display_bitenum
display_builtin_error
display_comm_error
display_comm_status
display_device_status
display_dynamics
display_dynamics2
display_message
display_message2
display_response_code
display_response_status
display_xmtr_status
edit_device_value

Builtin name
edit_device_value2
edit_local_value
edit_local_value2
get_acknowledgement2
get_acknowledgement
GET_DEV_VAR_VALUE
get_dev_var_value
GET_LOCAL_VAR_VALUE
get_local_var_value
Menu
MenuDisplay (version A)
MenuDisplay (version B)
PUT_MESSAGE
put_message
SELECT_FROM_LIST
select_from_list
select_from_menu
select_from_menu2

5.3 Communication builtins

The communication builtins allow methods to communicate with the device or to control the communication with the device. The list of communication builtins is listed in Table 8.

Table 8 – Communication builtins

Builtin name
abort_on_all_comm_errors
ABORT_ON_ALL_COMM_STATUS
ABORT_ON_ALL_DEVICE_STATUS
ABORT_ON_ALL_RESPONSE_CODES
abort_on_all_response_codes
ABORT_ON_COMM_ERROR
abort_on_comm_error
ABORT_ON_COMM_STATUS
ABORT_ON_DEVICE_STATUS
ABORT_ON_NO_DEVICE
ABORT_ON_RESPONSE_CODE
abort_on_response_code
ext_send_command
ext_send_command_trans
fail_on_all_comm_errors
fail_on_all_response_codes
fail_on_comm_error
fail_on_response_code

Builtin name
get_more_status
IGNORE_ALL_COMM_STATUS
IGNORE_ALL_DEVICE_STATUS
IGNORE_ALL_RESPONSE_CODES
IGNORE_COMM_ERROR
IGNORE_COMM_STATUS
IGNORE_DEVICE_STATUS
IGNORE_NO_DEVICE
IGNORE_RESPONSE_CODE
read_value
read_value2
ReadCommand
retry_on_all_comm_errors
RETRY_ON_ALL_COMM_STATUS
RETRY_ON_ALL_DEVICE_STATUS
RETRY_ON_ALL_RESPONSE_CODES
retry_on_all_response_codes
RETRY_ON_COMM_ERROR
retry_on_comm_error
RETRY_ON_COMM_STATUS
RETRY_ON_DEVICE_STATUS
RETRY_ON_NO_DEVICE
RETRY_ON_RESPONSE_CODE
retry_on_response_code
send
send_all_values
send_command
send_command_trans
send_trans
send_value
send_value2
SET_NUMBER_OF_RETRIES
WriteCommand
XMTR_ABORT_ON_ALL_COMM_STATUS
XMTR_ABORT_ON_ALL_DATA
XMTR_ABORT_ON_ALL_DEVICE_STATUS
XMTR_ABORT_ON_ALL_RESPONSE_CODES
XMTR_ABORT_ON_COMM_ERROR
XMTR_ABORT_ON_COMM_STATUS
XMTR_ABORT_ON_DATA
XMTR_ABORT_ON_DEVICE_STATUS
XMTR_ABORT_ON_NO_DEVICE
XMTR_ABORT_ON_RESPONSE_CODE
XMTR_IGNORE_ALL_COMM_STATUS

Builtin name
XMTR_IGNORE_ALL_DATA
XMTR_IGNORE_ALL_DEVICE_STATUS
XMTR_IGNORE_ALL_RESPONSE_CODES
XMTR_IGNORE_COMM_ERROR
XMTR_IGNORE_COMM_STATUS
XMTR_IGNORE_DATA
XMTR_IGNORE_DEVICE_STATUS
XMTR_IGNORE_NO_DEVICE
XMTR_IGNORE_RESPONSE_CODE
XMTR_RETRY_ON_ALL_COMM_STATUS
XMTR_RETRY_ON_ALL_DATA
XMTR_RETRY_ON_ALL_DEVICE_STATUS
XMTR_RETRY_ON_ALL_RESPONSE_CODE
XMTR_RETRY_ON_ALL_RESPONSE_CODES
XMTR_RETRY_ON_COMM_ERROR
XMTR_RETRY_ON_COMM_STATUS
XMTR_RETRY_ON_DATA
XMTR_RETRY_ON_DEVICE_STATUS
XMTR_RETRY_ON_NO_DEVICE
XMTR_RETRY_ON_RESPONSE_CODE

5.4 Action builtins

The action builtins shall only be used for actions called from a VARIABLE. They allow for accessing the value of the variable from which the action was called. The list of action builtins is listed in Table 9.

Table 9 – Action builtins

Reference	Builtin name	Allowed data type for action variable	Allowed actions from which the builtin can be called
6.4.1	fgetval	FLOAT, DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

Reference	Builtin name	Allowed data type for action variable	Allowed actions from which the builtin can be called
6.4.2	fsetval	FLOAT, DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.3	get_date	DATE_AND_TIME, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.4	get_double	DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.5	get_float	FLOAT	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.6	get_signed	INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.7	get_string	ASCII, BITSTRING, BIT_ENUMERATED, ENUMERATED, EUC, INDEX, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.8	get_unsigned	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, UNSIGNED_INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.9	igetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

Reference	Builtin name	Allowed data type for action variable	Allowed actions from which the builtin can be called
6.4.10	isetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.12	lgetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS
6.4.13	isetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS
6.4.14	put_date	DATE_AND_TIME, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.15	put_double	DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.16	put_float	FLOAT	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.17	put_signed	INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.18	put_string	ASCII, BITSTRING, BIT_ENUMERATED, ENUMERATED, EUC, INDEX, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS

Reference	Builtin name	Allowed data type for action variable	Allowed actions from which the builtin can be called
6.4.19	put_unsigned	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, UNSIGNED_INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.20	sgetval	ASCII, BITSTRING, EUC, PACKED_ASCII, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.21	ssetval	ASCII, BITSTRING, EUC, PACKED_ASCII, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

6 EDDL builtin library

6.1 General

Clause 6 describes builtin subroutines, which can be used in an EDD. The builtins support user or device interactions. Typical user interactions are, for example, user inputs of VARIABLE values or acknowledgments. Typical device interactions are, for example, send commands to the device or interpreting response codes and status bytes. This Clause 6 contains the library of builtins.

No other builtin functions are available and a compliant EDD shall only use the library functions defined in this specification. Requirements that all conforming EDD applications shall meet are defined in this specification. A conformant EDD application shall support all library functions found in this specifications that are applicable to the respective communication profile(s). A EDD application shall support all compliant EDDs without requiring any modifications to the compliant EDD and without any additional information beyond that found in the compliant EDD.

6.2 User interface builtins

6.2.1 Builtin ACKNOWLEDGE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display message and wait for key press

The builtin ACKNOWLEDGE will display the prompt and wait for acknowledgment from the user.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

ACKNOWLEDGE(prompt)

The lexical elements of the builtin ACKNOWLEDGE are shown in Table 10.

Table 10 – Builtin ACKNOWLEDGE

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	Message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	One of the return codes specified in Table 377.

6.2.2 Builtin acknowledge

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display message and wait for key press

The builtin acknowledge will display the prompt and wait for acknowledgment from the user.

NOTE 1 New Device Descriptions can use the builtin ACKNOWLEDGE, which supports direct referencing of VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 2 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 3 Some EDD applications will clear the display upon each call to this function.

Lexical structure

acknowledge(prompt, global_var_ids)

The lexical elements of the builtin acknowledge are shown in Table 11.

Table 11 – Builtin acknowledge

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	Message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	m	An array that contains unique identifiers of VARIABLE instances.
<return>	int	O	m	One of the return codes specified in Table 377.

6.2.3 Builtin DELAY

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display a message and delay

The builtin DELAY will display a prompt message and pause for the specified number of seconds. The delay time shall be a positive number.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

DELAY(delay_time, prompt)

The lexical elements of the builtin DELAY are shown in Table 12.

Table 12 – Builtin DELAY

Parameter name	Type	Direction	Usage	Description
delay_time	unsigned integer	I	m	The delay in number of seconds.
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.2.4 Builtin delay

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display a message and delay

The builtin delay will display a prompt message, and pause for the specified number of seconds. The delay time shall be a positive number.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 New Device Descriptions can use DELAY, which supports direct referencing of VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

Lexical structure

delay(delay_time, prompt, global_var_ids)

The lexical elements of the builtin delay are shown in Table 13.

Table 13 – Builtin delay

Parameter name	Type	Direction	Usage	Description
delay_time	unsigned integer	I	m	The delay in number of seconds.
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	m	An array which contains unique identifiers of VARIABLE instances.
<return>	VOID	—	—	—

6.2.5 Builtin delayfor

Communication profiles: FF, ISA100

Purpose: Display a message and wait the specified time

The builtin delayfor prints a string on the application display and waits for the allotted amount of time to expire.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
delayfor(duration, prompt, ids, indices, id_count)
```

The lexical elements of the builtin delayfor are shown in Table 14.

Table 14 – Builtin delayfor

Parameter name	Type	Direction	Usage	Description
duration	long	I	m	The delay in number of seconds. This shall be a positive number.
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of a VARIABLE instance used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array indices of the VARIABLEs used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long ids[2];
unsigned long indices[2];
ids[0] = ids[1] = ITEM_ID(PARAM.OPERATING_TEMP);
indices[0] = MEMBER_ID(HI_LIM);
indices[1] = MEMBER_ID(LO_LIM);
status = delayfor(5, "|en|Operating temperature limits - HI: %{0}
LO: %{1}\n", ids, indices, 2);
```

6.2.6 Builtin delayfor2

Communication profiles: FF, ISA100

Purpose: Display a message and wait the specified time

The builtin delayfor2 prints a string on the application display and waits for the allotted amount of time to expire.

VARIABLEs, method-variables, and labels for VARIABLEs may be embedded in the prompt string with formatting information. The blk_ids, blk_nums, ids, indices, and id_count parameters collectively specify the values to be embedded in the prompt.

The prompt parameter may contain more than one language translation as shown below.

```
"|en|Is limit correct?|de|Grenze richtig?"
```

Each element of the blk_ids, blk_nums, ids, and indices array specify a value to be embedded in the prompt. For example, in a case where there are three values embedded in a prompt, the values would be specified as follows:

```
blk_ids[0], blk_nums[0], ids[0], indices[0]
```

specifies the first value embedded in the prompt

```
blk_ids[1], blk_nums [1], ids[1], indices [1]
```

specifies the second value embedded in the prompt

```
blk_ids[2], blk_nums [2], ids[2], indices [2]
```

specifies the third value embedded in the prompt

Some of the builtin functions do not use the blk_ids and blk_nums parameters, while others do. When these parameters are used, the builtin function retrieves the values from the specified block. When these parameters are not used, the builtin function retrieves the values from the block that is scoped to the calling method.

Prompt strings may include format specifiers that specify values to be embedded in the prompt. A format specifier takes one of the following forms:

```
%format-specifier{array-index}
```

or

```
%L{array-index}
```

or

```
%format-specifier{local-variable-name}
```

where format-specifier specifies an optional printf format specifier, as defined in ISO/IEC 9899 plus the additions shown in Table 49. If a format specifier is not specified, the value's DISPLAY_FORMAT attribute is used. If a DISPLAY_FORMAT attribute is not specified, a default format is used based on the type of the value. array-index specifies an index in the blk_ids, blk_nums, ids, and indices array used to specify the value to be embedded in the prompt. local-variable-name specifies the name of the local variable to be embedded in the prompt. The %L form will embed the label of a variable, record member, or array element in the prompt, instead of a value.

The blk_ids parameter specifies the item ids of the blocks to be accessed, while the blk_nums parameter specifies the relative indices of the blocks to be accessed.

If an element of the ids parameter references a variable, the corresponding element of the indices parameter is ignored. If an element of the ids parameter references a record, the corresponding element of the indices parameter specifies a member of the record. If an element of the ids parameter references an array, the corresponding element of the indices parameter specifies an element of the array.

The elements of the blk_ids, ids, and indices parameters are typically specified using the ITEM_ID() and MEMBER_ID() builtin functions. The elements of the blk_nums parameter are typically specified using the get_block_instance_by_object_index() and get_block_instance_by_block_tag() builtin functions.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

The indices parameter may be passed as NULL if the ids parameter only references variables, i.e., it does not reference any records or arrays.

The id_count parameter shall be passed as zero if the blk_ids, blk_nums, and ids parameters are passed as NULL.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

delayfor2(duration, prompt, blk_ids, blk_nums, ids, indices, id_count)

The lexical elements of the builtin delayfor2 are shown in Table 15.

Table 15 – Builtin delayfor2

Parameter name	Type	Direction	Usage	Description
duration	long	l	m	The delay in number of seconds. This shall be a positive number.
prompt	DD_STRING	l	m	A message to be displayed. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	l	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	l	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	l	m	The item ids of VARIABLES, RECORDS, or ARRAYs used in the prompt.

Parameter name	Type	Direction	Usage	Description
indices	unsigned long[]	I	m	The member ids of RECORDs or array indices of ARRAYs used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long blk_ids[2];
unsigned long blk_nums [2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums [0] = blk_nums[1] = 0;
ids[0] = ids[1] = MEMBER_ID(OPERATING_TEMP);
indices[0] = MEMBER_ID(HI_LIM);
indices[1] = MEMBER_ID(LO_LIM);
status = delayfor2(5, "|en|Operating temperature limits - HI: %{0} LO:
%{1}\n", blk_ids, blk_nums, ids, indices, 2);

```

6.2.7 Builtin DISPLAY

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display the specified message on the screen

The builtin DISPLAY displays the specified message on the screen, continuously updating the dynamic variable values used in the string. The variable value will continue to be updated until operator acknowledgment is made.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

DISPLAY(prompt)

The lexical elements of the builtin DISPLAY are shown in Table 16.

Table 16 – Builtin DISPLAY

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.2.8 Builtin display

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display the specified message on the screen

The builtin display will display the specified message on the screen, continuously updating the dynamic VARIABLE values used in the string. The VARIABLE value will continue to be updated until operator acknowledgement is made.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 New Device Descriptions can use DISPLAY, which supports direct referencing of method-variables and VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

Lexical structure

`display(prompt, global_var_ids)`

The lexical elements of the builtin display are shown in Table 17.

Table 17 – Builtin display

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	m	An array which contains unique identifiers of VARIABLE instances.
<return>	VOID	—	—	—

6.2.9 Builtin display_bitenum

Communication profiles: CS, GPE, PB, PN

Purpose: Display enumeration string

The builtin display_bitenum will display a textual representation of a BIT_ENUMERATED VARIABLE and wait for acknowledgment from the user.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

`display_bitenum(bitenum_variable)`

The lexical elements of the builtin display_bitenum are shown in Table 18.

Table 18 – Builtin display_bitenum

Parameter name	Type	Direction	Usage	Description
bitenum_variable	Reference	I	m	The BIT_ENUMERATED VARIABLE that shall be displayed.
<return>	VOID	—	—	—

6.2.10 Builtin display_builtin_error

Communication profiles: FF, ISA100

Purpose: Display builtin error description and wait for acknowledge

The builtin display_builtin_error displays on the application display device the text associated with a specified builtin error. The builtin display does not return control to the method until the user acknowledges the message.

The error codes and their text descriptions are specified in Table 376.

This builtin provides a means of debugging calls to builtin routines, which return an error. It should never be used in a final production method, because the types of errors returned by builtins should never be received in a fully debugged method.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_builtin_error(error)

The lexical elements of the builtin display_builtin_error are shown in Table 19.

Table 19 – Builtin display_builtin_error

Parameter name	Type	Direction	Usage	Description
error	long	I	m	A code for a builtin error.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
if (status != BLTIN_SUCCESS)
{
    display_builtin_error(status);
}
```

6.2.11 Builtin display_comm_error

Communication profiles: FF, ISA100

Purpose: Display communication error description and wait for acknowledge

The builtin display_comm_error displays the text string associated with a communications error on the application display device.

NOTE 1 A communications error of zero is not considered a communications error.

NOTE 2 The builtin does not return control to the method until the user acknowledges the message.

Internally, this builtin calls the builtin get_comm_error_string to get the associated text string and the builtin get_acknowledgement to display the associated text string and wait for user acknowledgement.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary.

NOTE 3 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 4 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
display_comm_error(error)
```

The lexical elements of the builtin display_comm_error are shown in Table 20.

Table 20 – Builtin display_comm_error

Parameter name	Type	Direction	Usage	Description
error	unsigned long	I	m	A code for a communication error.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
if (status == BLTIN_FAIL_COMM_ERROR)
{
    status = display_comm_error(get_comm_error());
}
```

The example uses the builtin get_comm_error() builtin to get the communications error code used as the argument to this builtin.

6.2.12 Builtin display_comm_status

Communication profiles: HART

Purpose: Display comm status string

The builtin display_comm_status will display the description string(s) of the communication status VARIABLE designated by parameter 'comm_status_value'. The displayed string will remain on the screen until operator acknowledgement is made.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
display_comm_status(comm_status_value)
```

The lexical elements of the builtin display_comm_status are shown in Table 21.

Table 21 – Builtin display_comm_status

Parameter name	Type	Direction	Usage	Description
comm_status_value	int	I	m	A code for a communication error.
<return>	VOID	—	—	—

6.2.13 Builtin display_device_status

Communication profiles: HART

Purpose: Display device status string

The builtin display_device_status will display the description string(s) of the device status VARIABLE designated by parameter 'device_status_value'. The displayed string will remain on the screen until operator acknowledgement is made.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_device_status(device_status_value)

The lexical elements of the builtin display_device_status are shown in Table 22.

Table 22 – Builtin display_device_status

Parameter name	Type	Direction	Usage	Description
device_status_value	int	l	m	A code for the device status.
<return>	VOID	—	—	—

6.2.14 Builtin display_dynamics

Communication profiles: FF, ISA100

Purpose: Display and update VARIABLE values

The builtin display_dynamics continuously displays a message on the application output device. The builtin updates the values specified in the message until interrupted by the user. Control is not returned to the method until the user interrupts.

The primary purpose of the call is to display device value information that the user wants to observe over a period of time.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_dynamics(prompt, ids, indices, id_count)

The lexical elements of the builtin display_dynamics are shown in Table 23.

Table 23 – Builtin display_dynamics

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long ids[2];
unsigned long indices[2];
ids[0] = ITEM_ID(PARAM.OUT);
ids[1] = ITEM_ID(PARAM.BKCAL_IN);
indices[0] = indices[1] = MEMBER_ID(VALUE);
status = display_dynamics("|en|PID.OUT: %{0} PID.BKCAL_IN: %{1}, ids,
indices, 2);

```

6.2.15 Builtin display_dynamics2**Communication profiles: FF, ISA100****Purpose: Display and update VARIABLE values**

The builtin display_dynamics2 continuously displays a message on the application output device. The builtin updates the values specified in the message until interrupted by the user. Control is not returned to the method until the user interrupts. The primary purpose of the call is to display device value information that the user wants to observe over a period of time.

The blk_ids, blk_nums, ids, indices, and id_count parameters collectively specify the values to be embedded in the prompt. The blk_ids parameter specifies the item ids of the blocks to be accessed, while the blk_nums parameter specifies the relative indices of the blocks to be accessed.

If an element of the ids parameter references a variable, the corresponding element of the indices parameter is ignored. If an element of the ids parameter references a record, the corresponding element of the indices parameter specifies a member of the record. If an element of the ids parameter references an array, the corresponding element of the indices parameter specifies an element of the array.

The elements of the blk_ids, ids, and indices parameters are typically specified using the ITEM_ID() and MEMBER_ID() builtin functions. The elements of the blk_nums parameter are typically specified using the get_block_instance_by_object_index() and get_block_instance_by_block_tag() builtin functions.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

`display_dynamics2(prompt, blk_ids, blk_nums, ids, indices, id_count)`

The lexical elements of the builtin `display_dynamics2` are shown in Table 24.

Table 24 – Builtin `display_dynamics2`

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of VARIABLES, RECORDs, or ARRAYs used in the prompt.
indices	unsigned long[]	I	m	The member ids of RECORDs or array indices of ARRAYs used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long blk_ids[2];
unsigned long blk_nums [2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] =
ITEM_ID(_proportional_integral_derivative_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(OUT);
ids[1] = MEMBER_ID(BKCAL_IN);
indices[0] = indices[1] = MEMBER_ID(VALUE);
status = display_dynamics2("|en|PID.OUT: %{0} PID.BKCAL_IN: %{1}|", blk_ids,
blk_nums, ids, indices, 2);
```

6.2.16 Builtin `display_message`

Communication profiles: FF, ISA100

Purpose: Displays a message that requires no user acknowledgment

The builtin `display_message` displays a message on the applications output device. It returns control to the method without requiring user acknowledgment.

This call is used to inform the user of system status changes that require no action. The message should remain available for review at least until a subsequent call potentially replaces the message with a new one, or the method ends.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

`display_message(prompt, ids, indices, id_count)`

The lexical elements of the builtin `display_message` are shown in Table 25.

Table 25 – Builtin `display_message`

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long ids[1];
unsigned long indices[1];
status = display_message("|en|All clear\n", ids, indices, 0);
```

6.2.17 Builtin `display_message2`

Communication profiles: FF, ISA100

Purpose: Displays a message that requires no user acknowledgment

The builtin `display_message2` displays a message on the applications output device. It returns control to the method without requiring user acknowledgment. This call is used to inform the user of system status changes that require no action. The message should remain available for review at least until a subsequent call potentially replaces the message with a new one, or the method ends.

The `blk_ids` parameter specifies the item ids of the blocks to be accessed, while the `blockInstances` parameter specifies the relative indices of the blocks to be accessed.

If an element of the `ids` parameter references a variable, the corresponding element of the `indices` parameter is ignored. If an element of the `ids` parameter references a record, the corresponding element of the `indices` parameter specifies a member of the record. If an element of the `ids` parameter references an array, the corresponding element of the `indices` parameter specifies an element of the array.

The elements of the `blk_ids`, `ids`, and `indices` parameters are typically specified using the builtins `ITEM_ID` and `MEMBER_ID`. The elements of the `bl_nums` parameter are typically specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

The indices parameter may be passed as NULL if the ids parameter only references variables, i.e., it does not reference any records or arrays.

The count parameter shall be passed as zero if the blk_ids, blk_nums, and indices parameters are passed as NULL.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_message2(prompt, blk_ids, blk_nums, ids, indices, id_count)

The lexical elements of the builtin display_message2 are shown in Table 26.

Table 26 – Builtin display_message2

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of the VARIABLES, RECORDs, or ARRAYs used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long blk_ids[1];
unsigned long blk_nums[1];
unsigned long ids[1];
unsigned long indices[1];
blk_ids[0] = ITEM_ID(__analog_input_block);
blk_nums[0] = 0;
ids[0] = MEMBER_ID(MODE_BLK);
indices[0] = MEMBER_ID(TARGET);
status = display_message2("|en|Target Mode is %{0}\n", blk_ids, blk_nums,
ids, indices, 1);
```

6.2.18 Builtin display_response_code

Communication profiles: FF, ISA100

Purpose: Display response code description and wait for acknowledge

The builtin display_response_code displays the description of a response code for an item. The display builtin does not return control to the method until the user has acknowledged the message.

Internally, this builtin calls the builtin get_response_code_string to get the associated text string and the builtin get_acknowledgement to display the associated text string and wait for user acknowledgement.

A response code is a value representing an application-specific error condition. A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

The parameter id and member_id refer to a parameter, which may have a response code associated with it after it was last read or written. The parameter id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the variable in question.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_response_code(id, member_id, code)

The lexical elements of the builtin display_response_code are shown in Table 27.

Table 27 – Builtin display_response_code

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the item which generated the response code.
member_id	unsigned long	I	m	The member id of the item which generated the response code.
code	unsigned long	I	m	The response code as received from builtin get_response_code.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long id;
unsigned long member;
unsigned long code;
if (status == BLTIN_FAIL_RESPONSE_CODE)
{
    get_response_code(&code, &id, &member);
    status = display_response_code(id, member, code);
}
```

The example uses the builtin get_response_code() to get the response code returned on the previous communications request.

6.2.19 Builtin display_response_status

Communication profiles: HART

Purpose: Display response status string

The builtin display_response_status will display the string associated with the specified value of the response_code octet received from the specified COMMAND NUMBER. The displayed string will remain on the screen until operator acknowledgement is made.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_response_status(command_num, response_code_value)

The lexical elements of the builtin display_response_status are shown in Table 28.

Table 28 – Builtin display_response_status

Parameter name	Type	Direction	Usage	Description
command_num	Int	I	m	The NUMBER of the COMMAND from which the response status has been received.
response_code_value	int	I	m	The response code to display in a prompt.
<return>	VOID	—	—	—

6.2.20 Builtin display_xmtr_status

Communication profiles: HART

Purpose: Display transmitter status string

The builtin display_xmtr_status will display a description string associated with the bit indicated by parameter 'status_mask' in the VARIABLE designated by parameter 'bit_enum_var_name'. The string is displayed irrespective of whether the bit indicated by the status_mask is set or not. The displayed string will remain on the screen until operator acknowledgement is made.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

display_xmtr_status(bit_enum_var_name, status_mask)

The lexical elements of the builtin display_xmtr_status are shown in Table 29.

Table 29 – Builtin display_xmtr_status

Parameter name	Type	Direction	Usage	Description
bit_enum_var_name	int	l	m	A reference to a VARIABLE instance of type BIT_ENUMERATED.
status_mask	int	l	m	A bit mask indicating the bit to displayed.
<return>	VOID	—	—	—

6.2.21 Builtin edit_device_value**Communication profiles: FF, ISA100****Purpose: Displays a device value as defined in EDDL and lets the user modify it**

The builtin edit_device_value displays a prompt with a device value specified by the param_id and param_member_id. The user can edit the displayed value.

A device value may reside in the cache of a caching EDD application, and the value shall not be written to the device until a method terminates or the value is explicitly sent to the device by the method.

The application edits and verifies the values entered for the device value. This editing is based on VARIABLE arithmetic types defined in the original EDD for each VARIABLE. The EDDL attributes associated with VARIABLES may include

- HANDLING
- DISPLAY_FORMAT
- EDIT_FORMAT
- MIN_VALUE
- MAX_VALUE

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
edit_device_value(prompt, ids, indices, id_count, param_id,
param_member_id)
```

The lexical elements of the builtin edit_device_value are shown in Table 30.

Table 30 – Builtin edit_device_value

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	l	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	l	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	l	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	l	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
param_id	unsigned long	l	m	The item id of the VARIABLE to be modified.

Parameter name	Type	Direction	Usage	Description
param_member_id	unsigned long	I	m	The member id of the VARIABLE to be modified.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long id;
unsigned long ids[1];
unsigned long indices[1];
id = ITEM_ID(PARAM.HI_LIM);
status = edit_device_value("|en|Enter new HI_LIM value:\n", ids, indices,
0, id, 0);
status = send_value(id, 0);
```

6.2.22 Builtin edit_device_value2

Communication profiles: FF, ISA100

Purpose: Displays a device value as defined in EDDL and lets the user modify it

The builtin edit_device_value2 displays a prompt and allows the user to edit the variable, record member, or array element specified by the blk_id, blk_num, param_id and param_member_id parameters.

A device value may reside in the cache of a caching EDD application, and the value shall not be written to the device until a method terminates or the value is explicitly sent to the device by the method.

The blk_ids, blk_nums, ids, indices, and count parameters collectively specify the values to be embedded in the prompt.

The application edits and verifies the values entered for the device value. This editing is based on attributes of the variable, record member, or array element defined in the original EDD for each VARIABLE. The EDDL attributes associated with VARIABLES may include:

- HANDLING
- DISPLAY_FORMAT
- EDIT_FORMAT
- MIN_VALUE
- MAX_VALUE

The blk_ids parameter specifies the item ids of the blocks to be accessed, while the blk_nums parameter specifies the relative indices of the blocks to be accessed.

If an element of the ids parameter references a variable, the corresponding element of the indices parameter is ignored. If an element of the ids parameter references a record, the corresponding element of the indices parameter specifies a member of the record. If an element of the ids parameter references an array, the corresponding element of the indices parameter specifies an element of the array.

The elements of the blk_ids, ids, and indices parameters are typically specified using the builtin ITEM_ID and MEMBER_ID. The elements of the blk_nums parameter are typically specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

The indices parameter may be passed as NULL if the ids parameter only references variables, i.e., it does not reference any records or arrays.

The count parameter shall be passed as zero if the blk_ids, blk_nums, and ids parameters are passed as NULL.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
edit_device_value2(prompt, blk_ids, blk_nums, ids, indices, id_count,
blk_id, blk_num, param_id, param_member_id)
```

The lexical elements of the builtin edit_device_value2 are shown in Table 31.

Table 31 – Builtin edit_device_value2

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of the VARIABLES, RECORDs, or ARRAYs used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the record members and/or array elements used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of elements in the blk_ids, blk_nums, ids and indices arrays. Set to zero if the arrays for ids and indices are NULL.
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to be modified.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to be modified.
param_id	unsigned long	I	m	The item id of the VARIABLE to be modified.
param_member_id	unsigned long	I	m	The member id of the VARIABLE to be modified.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long blk_ids[1];
unsigned long blk_nums[1];
unsigned long ids[1];
unsigned long indices[1];
unsigned long blk_id;
unsigned long blk_num;
unsigned long param_id;
blk_id = ITEM_ID(__analog_input_block);
blk_num = 0;
param_id = MEMBER_ID(HI_LIM);
status = edit_device_value2("\n|Enter new HI_LIM value:\n", blk_ids,
blk_nums, ids, indices, 0, blk_id, blk_num, param_id, 0);

```

6.2.23 Builtin edit_local_value**Communication profiles: FF, ISA100****Purpose: Displays a method-variable and lets the user modify it**

The builtin edit_local_value displays a prompt with the method-variable name specified by the parameter local_var. The local_var parameter is a null-terminated C string of characters. The user can edit the displayed value of the method-variable.

The EDD application is responsible for preventing invalid values from being returned by performing checks based on the method-variable type only. The method shall perform range checking on returned values.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
edit_local_value(prompt, ids, indices, id_count, local_var)
```

The lexical elements of the builtin edit_local_value are shown in Table 32.

Table 32 – Builtin edit_local_value

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
local_var	char[]	I	m	The name of the local VARIABLE that may be modified.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long ids[1];
unsigned long indices[1];
float offset;
offset = 1.0;
status = edit_local_value("|en|Please enter offset...\n", ids, indices, 0,
"offset");
```

6.2.24 Builtin edit_local_value2**Communication profiles: FF, ISA100****Purpose: Displays a method-variable and lets the user modify it**

The builtin `edit_local_value2` displays a prompt with the method-variable name specified by the parameter `local_var`. The `local_var` parameter is a null-terminated C string of characters. The user can edit the displayed value of the method-variable.

The `blk_ids`, `blk_nums`, `ids`, `indices`, and `count` parameters collectively specify the values to be embedded in the prompt.

The EDD application is responsible for preventing invalid values from being returned by performing checks based on the method-variable type only. The method shall perform range checking on returned values.

The `blk_ids` parameter specifies the item `ids` of the blocks to be accessed, while the `blk_nums` parameter specifies the relative indices of the blocks to be accessed.

If an element of the `ids` parameter references a variable, the corresponding element of the `indices` parameter is ignored. If an element of the `ids` parameter references a record, the corresponding element of the `indices` parameter specifies a member of the record. If an element of the `ids` parameter references an array, the corresponding element of the `indices` parameter specifies an element of the array.

The elements of the `blk_ids`, `ids`, and `indices` parameters are typically specified using the builtin `ITEM_ID` and `MEMBER_ID`. The elements of the `blk_nums` parameter are typically specified using the builtin `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The `blk_ids`, `blk_nums`, and `ids` parameters may be passed as `NULL` if there are no values embedded in the prompt. If any of these parameters are passed as `NULL`, they all shall be passed as `NULL`.

The `indices` parameter may be passed as `NULL` if the `ids` parameter only references variables, i.e., it does not reference any records or arrays.

The `count` parameter shall be passed as zero if the `blk_ids`, `blk_nums`, and `ids` parameters are passed as `NULL`.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
edit_local_value2(prompt, blk_ids, blk_nums, ids, indices, id_count,
local_var)
```

The lexical elements of the builtin `edit_local_value2` are shown in Table 33.

Table 33 – Builtin edit_local_value2

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array index of the VARIABLES used in the prompt message. Set to zero if there is no member id or array index.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
local_var	char[]	I	m	The name of the local VARIABLE that may be modified.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
float offset;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[0];
offset = 1.0;
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(LOW_LIMIT);
ids[1] = MEMBER_ID(HIGH_LIMIT);
indices[0] = indices[1] = 0;
status = edit_local_value2("|en|Please enter offset between %{0} and
%{1}\n", blk_ids, blk_nums, ids, indices, 2, "offset");

```

6.2.25 Builtin get_acknowledgement**Communication profiles: FF, ISA100****Purpose: Displays a message that requires user acknowledgment**

The builtin get_acknowledgement displays the provided prompt message on the application display device and returns to the method after the user acknowledges the message.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
get_acknowledgement(prompt, ids, indices, id_count)
```

The lexical elements of the builtin get_acknowledgement are shown in Table 34.

Table 34 – Builtin get_acknowledgement

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array indices of the VARIABLES used in the prompt message. Set to zero if there is no member id or array indices.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long ids[1];
unsigned long indices[1];
status = get_acknowledgement("|en|W A R N\n loop should be removed from
auto mode\n", ids, indices, 0,);

```

6.2.26 Builtin get_acknowledgement2**Communication profiles: FF, ISA100****Purpose: Displays a message that requires user acknowledgment**

The builtin get_acknowledgement2 displays the provided prompt message on the application display device and returns to the method after the user acknowledges the message. The blk_ids, blk_nums, ids, indices, and count parameters collectively specify the values to be embedded in the prompt.

The blk_ids parameter specifies the item ids of the blocks to be accessed, while the blk_nums parameter specifies the relative indices of the blocks to be accessed.

If an element of the ids parameter references a variable, the corresponding element of the indices parameter is ignored. If an element of the ids parameter references a record, the corresponding element of the indices parameter specifies a member of the record. If an element of the ids parameter references an array, the corresponding element of the indices parameter specifies an element of the array.

The elements of the blk_ids, ids, and indices parameters are typically specified using the builtins ITEM_ID and MEMBER_ID. The elements of the blk_nums parameter are typically specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

The indices parameter may be passed as NULL if the ids parameter only references variables, i.e., it does not reference any records or arrays.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
get_acknowledgement2(prompt, blk_ids, blk_nums, ids, indices, id_count)
```

The lexical elements of the builtin `get_acknowledgement2` are shown in Table 35.

Table 35 – Builtin `get_acknowledgement2`

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of the VARIABLES, RECORD, or ARRAYs used in the prompt message.
indices	unsigned long[]	I	m	The member ids or array indices of the members and/or array elements used in the prompt message. Set to zero if there is no member id or array indices.
id_count	long	I	m	The number of array elements in the arrays for ids and indices. Set to zero if the arrays for ids and indices are NULL.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(LOW_LIMIT);
ids[1] = MEMBER_ID(HIGH_LIMIT);
indices[0] = indices[1] = 0;
status = get_acknowledgement2("Setting LOW = %{0} and HIGH = %{1}\n",
blk_ids, blk_nums, ids, indices, 2);
```

6.2.27 Builtin GET_DEV_VAR_VALUE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display the specified prompt message and allow the user to edit the value of a VARIABLE

The builtin `GET_DEV_VAR_VALUE` will display the specified prompt message, and allow the user to edit the value of a VARIABLE. The edited copy of the VARIABLE value will be updated when the new value is entered, but will not be sent to the device. This shall be done explicitly using one of the send command functions.

`PRE_EDIT_ACTIONS` of the VARIABLE will be called prior to prompting the new value. `POST_EDIT_ACTIONS` will be called after a new value is entered.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

GET_DEV_VAR_VALUE(prompt, device_var_name)

The lexical elements of the builtin GET_DEV_VAR_VALUE are shown in Table 36.

Table 36 – Builtin GET_DEV_VAR_VALUE

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
device_var_name	reference	I	m	The referenced name of the VARIABLE, which can be edited.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the variable was successfully modified, or BI_ERROR if an error occurred entering the new value or accessing the specified variable.

6.2.28 Builtin get_dev_var_value

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display the specified prompt message and allow the user to edit⁵ the value of a VARIABLE

The builtin get_dev_var_value will display the specified prompt message and allow the user to edit the value of a VARIABLE. The edited copy of the VARIABLE value will be updated when the new value is entered but will not be sent to the device. This shall be done explicitly using one of the send command functions.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

PRE_EDIT_ACTIONS of the VARIABLE will be called prior to prompting the new value. POST_EDIT_ACTIONS will be called after a new value is entered.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 New Device Descriptions can use GET_DEV_VAR_VALUE, which supports direct referencing of method-variables and VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

Lexical structure

get_dev_var_value(prompt, global_var_ids, device_var_name)

The lexical elements of the builtin get_dev_var_value are shown in Table 37.

⁵ The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

Table 37 – Builtin get_dev_var_value

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	m	An array which contains unique identifiers of VARIABLE instance.
device_var_name	reference	I	m	The name of the VARIABLE, which can be edited.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the variable was successfully modified, or BI_ERROR if an error occurred entering the new value or accessing the specified variable.

6.2.29 Builtin GET_LOCAL_VAR_VALUE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: display the specified prompt message and allow the user to edit the value of a method-variable

The builtin GET_LOCAL_VAR_VALUE will display the specified prompt message and allow the user to edit the value of a method-variable.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

GET_LOCAL_VAR_VALUE(prompt, local_var_name)

The lexical elements of the builtin GET_LOCAL_VAR_VALUE are shown in Table 38.

Table 38 – Builtin GET_LOCAL_VAR_VALUE

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
local_var_name	referenced	I	m	The name of the method-variable to edit.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the variable was successfully modified, and BI_ERROR if an error occurred entering the new value or accessing the specified variable.

6.2.30 Builtin get_local_var_value

Communication profiles: CS, GPE, HART, PB, PN

Purpose: display the specified prompt message and allow the user to edit the value of a method-variable

The builtin get_local_var_value will display the specified prompt message and allow the user to edit the value of a method-variable.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 New Device Descriptions can use GET_LOCAL_VAR_VALUE, which supports direct referencing of method-variables and VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

Lexical structure

```
get_local_var_value(prompt, global_var_ids, local_var_name)
```

The lexical elements of the builtin get_local_var_value are shown in Table 39.

Table 39 – Builtin get_local_var_value

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	o	An array which contains unique identifiers of VARIABLE instance.
local_var_name	referenced	I	o	The name of the local VARIABLE to edit.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the variable was successfully modified, and BI_ERROR if an error occurred entering the new value or accessing the specified variable.

6.2.31 Builtin Menu

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display a menu to the user

The builtin Menu will display a MENU and wait for acknowledgment from the user using a single option (i.e., "OK").

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Communication profiles: HART

The MENU referenced from this builtin shall not contain other methods or edit displays as menu items.

The MENU shall be treated as a MENU with STYLE DIALOG regardless of the actual STYLE of the MENU and any sub menu items of type MENU shall be of style GROUP.

Communication profiles: CS, GPE, PB, PN

The MENU shall be treated as a MENU with STYLE DIALOG regardless of the actual STYLE of the MENU.

Lexical structure

```
Menu (menu)
```

If the EDD application is unable to display the menu, an error will be returned.

The lexical elements of the builtin Menu are shown in Table 40.

Table 40 – Builtin Menu

Parameter name	Type	Direction	Usage	Description
menu	reference	I	m	The MENU to be displayed.
<return>	long	O	m	Communication profiles: CS, GPE, HART, PB, PN One of the return codes specified in Table 377. BI_SUCCESS if the function was successful and BI_ERROR if the function failed.

6.2.32 Builtin MenuDisplay (version A)

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Display a menu to the user and a supplied list of options for the user to select

The builtin MenuDisplay will display a MENU and wait for a user acknowledgement by selecting one option from the supplied option list. In addition to this, to avoid operator confusion, option_list should not use phrases like "Abort" and "Cancel". Once a selection is made, the option position in the option list is returned. The position in the list is zero-based. For example, if the provided list contains two items and the first item is selected, this function will return a 0. If the second item is selected, a 1 is returned.

The options are passed to the function as a single string with semicolon separators between the options. If no options are listed, the function will return an error.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return. The cancel mechanism to abort the method shall be provided by the EDD application.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 Some limited display EDD applications are only able to display 4 buttons. Thus when three developer-defined buttons are included (plus the standard Abort mechanism), it is possible that the "Help" button is no longer available.

Communication profiles: HART

No more than 3 user-defined options are allowed, excluding the cancel mechanism that shall be provided by the EDD application.

The MENU shall be treated as a MENU with STYLE DIALOG regardless of the actual STYLE of the MENU and any sub menu items of type MENU shall be of style GROUP. The MENU referenced from this builtin shall not contain other methods or edit displays as menus items.

Communication profiles: CS, GPE, PB, PN

The MENU shall be treated as a MENU with STYLE DIALOG regardless of the actual STYLE of the MENU.

Lexical structure

MenuDisplay(menu, options, selection)

If the EDD application is unable to display the menu, an error will be returned.

The lexical elements of the builtin MenuDisplay are shown in Table 41.

Table 41 – Builtin MenuDisplay

Parameter name	Type	Direction	Usage	Description
menu	reference	I	m	The MENU to be displayed.
options	DD_STRING	I	m	The option string, consisting of separated text, that defines a set of options presented to the user. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
selection	long*	O	m	The zero-based selected option (see options parameter).
<return>	long	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the function was successful and BI_ERROR if the function failed. If BI_ERROR is returned, the value of selection is not defined.

6.2.33 Builtin MenuDisplay (version B)

Communication profiles: FF, ISA100

Purpose: Display a menu to the user and a supplied list of options for the user to select

The builtin MenuDisplay will display a MENU and wait for a user acknowledgement by selecting one option from the supplied option list. In addition to this, to avoid operator confusion, option_list should not use phrases like "Abort" and "Cancel". Once a selection is made, the option position in the option list is returned. The position in the list is zero-based. For example, if the provided list contains two items and the first item is selected, this function will return a 0. If the second item is selected, a 1 is returned.

The options are passed to the function as a single string with semicolon separators between the options. If no options are listed, the Function will return an error.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 Some limited display EDD applications are only able to display 4 buttons. Thus when three developer-defined buttons are included (plus the standard Abort mechanism), it is possible that the "Help" button is no longer available.

No more than 3 user-defined options are allowed, excluding the cancel mechanism that shall be provided by the EDD application.

The MENU shall be treated as a MENU with STYLE DIALOG regardless of the actual STYLE of the MENU. The MENU referenced from this builtin shall not contain other methods or edit displays as menu items.

Lexical structure

MenuDisplay(menu, options, selection)

If the EDD application is unable to display the menu, an error will be returned.

The lexical elements of the builtin MenuDisplay are shown in Table 42.

Table 42 – Builtin MenuDisplay

Parameter name	Type	Direction	Usage	Description
menu	unsigned long	I	m	The item id of the MENU to be displayed.
options	DD_STRING	I	m	The option string, consisting of separated text, that defines a set of options presented to the user. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
selection	long*	O	m	The zero-based selected option (see options parameter).
<return>	long	O	m	One of the return codes specified in Table 376 .

Example

```

long select;
long step;
select = 0;
step = 0;
do
{
    switch (step)
    {
        case 0:
            MenuDisplay(ITEM_ID(wizard_step0), "NEXT", &select);
            step++;
            break;
        case 1:
            MenuDisplay(ITEM_ID(wizard_step1), "BACK;NEXT", &select);
            if (select == 0)
                step = 0;
            if (select == 1)
                step = 2;
            break;
        case 2:
            MenuDisplay(ITEM_ID(wizard_step2), "FINISH", &select);
            step++;
            break;
        default:
            break;
    }
} while (step < 3);

```

6.2.34 Builtin PUT_MESSAGE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Displays the specified message on the screen

The builtin PUT_MESSAGE will display the specified message on the screen. Any embedded variable references will be expanded to the current value of the variable. ENUMERATED variables should be displayed according to their textual representation when defined. When an ENUMERATED value is undefined, they should be displayed according to their numerical representation.

Communication profiles: HART

BIT_ENUMERATED variables shall be displayed as integers in hexadecimal format.

Communication profiles: PB, PN, CS, GPE

BIT_ENUMERATED variables should either be displayed numerically as a hexadecimal integer, or according to the textual representation containing all bits set to "1".

No operator input is accepted by PUT_MESSAGE and the user is not offered the opportunity to abort the method.

Method execution continues immediately i.e., does not block the method execution. The message string may contain embedded references to any combination of method-variables and VARIABLE values. The message should remain available for review at least until a subsequent call potentially replaces the message with a new one, or the method ends.

References to variables may be embedded on the message string allowing their value to be displayed using the syntax specified in 6.2.40.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

PUT_MESSAGE (message)

The lexical elements of the builtin PUT_MESSAGE are shown in Table 43.

Table 43 – Builtin PUT_MESSAGE

Parameter name	Type	Direction	Usage	Description
message	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

Example

```
int sum;
PUT_MESSAGE("The sum is: %[5.2f]{sum}");
```

The example displays the value of the variable “sum”.

```
PUT_MESSAGE("The loop current is %[4.2f,D]{loop_current} mA");
```

The example displays and continuously updates the value of the variable “loop_current”.

```
PUT_MESSAGE("%[L]{flow_value} = %{flow_value}");
```

The example displays the label and the value of the variable “flow_value”.

6.2.35 Builtin put_message

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Displays the specified message on the screen

The builtin put_message will display the specified message on the screen. Any embedded variable references will be expanded to the current value of the variable. ENUMERATED variables should be displayed according to their textual when defined. When an ENUMERATED value is undefined, they should be displayed according to their numerical representation.

Communication profiles: HART

BIT_ENUMERATED variables shall be displayed as integers in hexadecimal format.

Communication profiles: PB, PN, CS, GPE

BIT_ENUMERATED variables should either be displayed numerically as a hexadecimal integer, or according to the textual representation containing all bits set to "1".

No operator input is accepted by `put_message` and the user is not offered the opportunity to abort the method. Method execution continues immediately i.e., does not block the method execution. The message should remain available for review at least until a subsequent call potentially replaces the message with a new one, or the method ends.

Any number of identifiers may be accessed by the string, but the array entries accessed shall contain valid variable identifiers. The message string may contain references to any combination of method-variables and VARIABLE values. References to method-variables shall be referenced by name, and references to VARIABLES shall be referenced by `global_var_id`.

The following syntax is used to reference a VARIABLE from the list of VARIABLES given in the parameter `global_var_ids`:

`%{global-var-ids-array-index}`

or

`%[format]{global-var-ids-array-index}`

The optional format supports all fprintf format strings the programming language-C (see ISO/IEC 9899:1999) plus the additions shown in Table 49. If the format string is not included then the default formatting specified in IEC 61804-3 shall be used.

NOTE 1 While not recommended for new DDs, (lowercase) `put_message` is supported (for backward compatibility).

NOTE 2 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 3 Some EDD applications will clear the display upon each call to this function.

Lexical structure

`put_message(message, global_var_ids)`

The lexical elements of the builtin `put_message` are shown in Table 44.

Table 44 – Builtin `put_message`

Parameter name	Type	Direction	Usage	Description
message	DD_STRING	I	m	A message to be displayed. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	int[]	I	m	An array which contains unique identifiers of VARIABLE instance.
<return>	VOID	—	—	—

Example

```
int ids[2];
ids[0] = VARID(pressure_units);
ids[1] = VARID(pressure_value);
put_message("Pressure value = {%1} {%0}",ids);
```

The example gets the id of the variables “pressure_value” and “pressure_units” and passes it to the builtin put_message for display.

6.2.36 Builtin SELECT_FROM_LIST

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Allow user to select one of multiple options

The builtin SELECT_FROM_LIST will display a prompt and a supplied list of options for the user to select from. The options are passed to the function as a single string with semicolon separators between the options. Once a selection is made, the position of that option in the option list is returned. The position in the list is zero-based. For example, if the list provided contains two items and if the first item is selected, this function will return 0. If the second item is selected, a 1 is returned. There shall be at least two options in the select list.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

SELECT_FROM_LIST(prompt, option_list)

The lexical elements of the builtin SELECT_FROM_LIST are shown in Table 45.

Table 45 – Builtin SELECT_FROM_LIST

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
option_list	DD_STRING	I	m	The option string, consisting of separated text, that defines the menu items. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	The result of the selection as position in the option list (zero based).

Example

```
int result;
result = SELECT_FROM_LIST("Is this correct?", "Yes;No");
if (result == 0) { ... } // user selected "yes"
else { ... }
```

This function would display the prompt "Is this correct?" and the two options "Yes" and "No". If "Yes" is selected, a 0 is returned and the code in the if statement is executed. If "No" is selected, a 1 is returned and the code in the else statement is executed.

6.2.37 Builtin select_from_list

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Allow user to select one of multiple options

The builtin select_from_list will display a prompt and a supplied list of options for the user to select from. Once a selection is made, the position of that option in the option list is returned. The position in the list is zero-based. For example, if the list provided contains two items and if the first item is selected, this function will return a 0. If the second item is selected, a 1 is returned.

The user has the opportunity to abort the method while method execution is halted waiting for this function to return.

The options are passed to the function as a single string with semicolon separators between the options. There shall be at least two options in the select list.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

NOTE 3 New Device Descriptions can use SELECT_FROM_LIST, which supports direct referencing of method-variables and VARIABLES. The lower case version of this function is maintained for backwards compatibility only.

Lexical structure

```
select_from_list(prompt, global_var_ids, option_list)
```

The lexical elements of the builtin select_from_list are shown in Table 46.

Table 46 – Builtin select_from_list

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.35 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
global_var_ids	Int[]	I	m	An array which contains unique identifiers of VARIABLE instances.
option_list	DD_STRING	I	m	The option string, consisting of semicolon separated text, that defines the menu items. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	The result of the selection as position (zero based).

6.2.38 Builtin select_from_menu

Communication profiles: FF, ISA100

Purpose: Displays a menu of choices and allows the user to select from them

The builtin select_from_menu displays a list of options or menu items for the user to select from and retrieves the selection number. When the user has made a choice, it is returned to the calling method. Selection numbers start at 1 and range up to the number of options specified by the parameter options.

The option string defines the menu from which the user selects. It is a string consisting of the selection options; each option is separated from the other options with a semi-colon. Each option may have language codes embedded in it. The first menu item shall be the default, i.e., it is pre-selected.

The application displays the strings in accordance with the following guidelines:

- the menu items appear on the screen in the same order as listed in the string;
- the prompt string appears at the top of the menu display on the screen;
- the prompt string is not selectable;
- the menu is presented vertically.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

`select_from_menu(prompt, ids, indices, id_count, options, selection)`

The lexical elements of the builtin `select_from_menu` are shown in Table 47.

Table 47 – Builtin `select_from_menu`

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or indices of the VARIABLES used in the prompt message.
id_count	long	I	m	The number of REFERENCE_ARRAY elements in the REFERENCE_ARRAY for ids and indices. Set to zero if the REFERENCE_ARRAY for ids and indices are zero.
options	DD_STRING	I	m	The option string, consisting of separated text, that defines the menu items. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
selection	long	O	m	The result of the selection as position (zero based).
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long paramIds[1];
unsigned long memberIds[1];
int choice;
float new_value;

status = select_from_menu("|en|Is limit correct?|de|Grenze richtig?",
paramIds, memberIds, 0, "|en|Yes|de|Ja;|en|No|de|Nein", &choice);

switch (choice)
{
case 1:
    status = put_float_value(MEMBER_ID(HI_LIM), new_value);
    status = send_all_values();
    break;

default:
    break;
}

```

6.2.39 Builtin select_from_menu2

Communication profiles: FF, ISA100

Purpose: Displays a menu of choices and allows the user to select from them

The builtin select_from_menu2 displays a list of options or menu items for the user to select from and retrieves the selection number. When the user has made a choice it is returned to the calling method. Selection numbers start at 1 and range up to the number of options specified by the parameter options.

The blk_ids, blk_nums, ids, indices, and id_count parameters collectively specify the values to be embedded in the prompt.

The blk_ids parameter specifies the item ids of the blocks to be accessed, while the blk_nums parameter specifies the relative indices of the blocks to be accessed.

If an element of the ids parameter references a variable, the corresponding element of the indices parameter is ignored. If an element of the ids parameter references a record, the corresponding element of the indices parameter specifies a member of the record. If an element of the ids parameter references an array, the corresponding element of the indices parameter specifies an element of the array.

The elements of the blk_ids, ids, and indices parameters are typically specified using the builtins ITEM_ID and MEMBER_ID. The elements of the blk_nums parameter are typically specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The blk_ids, blk_nums, and ids parameters may be passed as NULL if there are no values embedded in the prompt. If any of these parameters are passed as NULL, they all shall be passed as NULL.

The indices parameter may be passed as NULL if the ids parameter only references variables, i.e., it does not reference any records or arrays.

The id_count parameter shall be passed as zero if the blk_ids, blk_nums, and ids parameters are passed as NULL.

The options string defines the menu from which the user selects. It is a string consisting of the selection options; each option is separated from the other options with a semi-colon. Each option may have language codes embedded in it. The first menu item shall be the default, i.e., it is pre-selected.

The application displays the strings in accordance with the following guidelines:

- the menu items appear on the screen in the same order as listed in the string;
- the prompt string appears at the top of the menu display on the screen;
- the prompt string is not selectable;
- the menu is presented vertically.

NOTE 1 Previously displayed messages are not guaranteed to remain on the screen.

NOTE 2 Some EDD applications will clear the display upon each call to this function.

Lexical structure

```
select_from_menu2(prompt, blk_ids, blk_nums, ids, indices, id_count,
options, selection)
```

The lexical elements of the builtin select_from_menu2 are shown in Table 48.

Table 48 – Builtin select_from_menu2

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.6 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
blk_ids	unsigned long[]	I	m	The item ids of the BLOCK_A instances used in the prompt message.
blk_nums	unsigned long[]	I	m	The occurrence numbers of the BLOCK_A instances used in the prompt message.
ids	unsigned long[]	I	m	The item ids of the VARIABLES used in the prompt message.
indices	unsigned long[]	I	m	The member ids or indices of the VARIABLES used in the prompt message.
id_count	long	I	m	The number of REFERENCE_ARRAY elements in the REFERENCE_ARRAY for ids and indices. Set to zero if the REFERENCE_ARRAY for ids and indices are zero.
options	DD_STRING	I	m	The option string, consisting of separated text, that defines the menu items. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
selection	long	O	m	The result of the selection as position (zero based).
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[2];
long choice;

blk_ids[0] = ITEM_ID(analog_input_block);
blk_nums[0] = 0;
ids[0] = MEMBER_ID(MODE_BLK);
indices[0] = MEMBER_ID(ACTUAL);

blk_ids[1] = ITEM_ID(__analog_input_block);
blk_nums[1] = 1;
ids[1] = MEMBER_ID(MODE_BLK);
indices[1] = MEMBER_ID(ACTUAL);

status = select_from_menu2("AI1 Mode: %{0}, AI2 Mode: %{1}\nIs this
correct?", blk_ids, blk_nums, ids, indices, 2, "Yes;No" &choice);

```

6.2.40 Prompt string formats

References to VARIABLES may be embedded on the prompt string allowing their value to be displayed using the following syntax:

```
%{variable-reference}
```

or

```
%[format]{variable-reference}
```

The variable-reference is the identifier of a method-variable or a VARIABLE. Method-variable and VARIABLES may be mixed in the same message. The optional format supports all fprintf format strings in the programming language-C (see ISO/IEC 9899:1999) plus the additions shown in Table 49. If the format string is not included then the default formatting specified in IEC 61804-3 shall be used.

Table 49 – Format options

Format	Description
D	The referenced item is of CLASS DYNAMIC and the display shall be updated continuously while the method execution is blocked by the builtin function. Items of CLASS DYNAMIC that are referenced without the D option shall not be updated dynamically. Items that are not of CLASS DYNAMIC that are referenced with the D option shall not be displayed dynamically. Non-blocking UI builtins (e.g. PUT_MESSAGE and put_message) will display a snapshot of the dynamic value only once. If a display format is given in addition to the "D" specifier, the D specifier shall be separated from the format specifier by a comma. Example: PUT_MESSAGE ("The variable with label '%L'{myVar}' has dynamic value: %[3.1f,D]{myVar}");
L	The LABEL of the referenced item shall be displayed. Example: PUT_MESSAGE ("The label of my variable is: %[L]{myVar}");
U	The unit of the referenced item shall be displayed. Example: PUT_MESSAGE ("The units of my variable is: %[U]{myVar}");

6.3 Communication builtins

6.3.1 Builtin abort_on_all_comm_errors

Communication profiles: FF, ISA100

Purpose: Set to abort for all communication errors

The builtin abort_on_all_comm_errors sets the method to abort upon receiving any communication error.

When a method is started, it does not abort on a communication error until abort_on_all_comm_errors is called. This frees a method from having to handle any communication error codes by aborting the method if it receives any communications error.

After this call, any builtin call which makes a communication request that returns an error will cause the method to abort and execution of its abort method list to be started. Communications error codes are non-zero and are EDD application dependent (each EDD application defines its own codes).

Lexical structure

abort_on_all_comm_errors(VOID)

The lexical element of the builtin abort_on_all_comm_errors is shown in Table 50.

Table 50 – Builtin abort_on_all_comm_errors

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
abort_on_all_comm_errors();
```

6.3.2 Builtin ABORT_ON_ALL_COMM_STATUS**Communication profiles:** HART**Purpose:** Set to abort for all communications status

The builtin ABORT_ON_ALL_COMM_STATUS will set all of the bits in the comm status abort mask. This will cause the system to abort the current method if the device returns any comm status value. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set to 1.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks, and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

```
ABORT_ON_ALL_COMM_STATUS (VOID)
```

The lexical element of the builtin ABORT_ON_ALL_COMM_STATUS is shown in Table 51.

Table 51 – Builtin ABORT_ON_ALL_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.3 Builtin ABORT_ON_ALL_DEVICE_STATUS**Communication profiles:** HART**Purpose:** Set to abort for all device status

The builtin ABORT_ON_ALL_DEVICE_STATUS will set all of the bits in the device status abort mask. This will cause the system to abort the current method if the device returns any non-zero device status value. The first two bytes of every slave response contains status information. The first byte returned indicates response code status when the most significant bit (bit 7) is 0. The second byte returned, device status, is only meaningful if the first returns response code status. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

```
ABORT_ON_ALL_DEVICE_STATUS (VOID)
```


The lexical element of the builtin ABORT_ON_ALL_DEVICE_STATUS is shown in Table 52.

Table 52 – Builtin ABORT_ON_ALL_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.4 Builtin ABORT_ON_ALL_RESPONSE_CODES

Communication profiles: HART

Purpose: Set to abort for all response codes

The builtin ABORT_ON_ALL_RESPONSE_CODES will set all of the bits in the response code abort mask. This will cause the system to abort the current method if the device returns any non-zero response code value.

The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send command function for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

NOTE Delayed responses and busies are Data-Link level codes and are not returned to Methods. I.e., the lower levels of the communications stack or the Remote I/O manage/react to the Response Codes. Depending on the end device being communicated with, a METHOD can see a DR_DEAD (the DR failed).

Lexical structure

ABORT_ON_ALL_RESPONSE_CODES (VOID)

The lexical element of the builtin ABORT_ON_ALL_RESPONSE_CODES is shown in Table 53.

Table 53 – Builtin ABORT_ON_ALL_RESPONSE_CODES

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.5 Builtin abort_on_all_response_codes

Communication profiles: FF, ISA100

Purpose: Set to abort for all response codes

The builtin abort_on_all_response_codes causes the method to abort if it receives any non-zero response code, after calling a builtin, which results in a request to a device. This builtin frees a method from having to handle any error response codes. When a method is started, it does not abort on any response code until this builtin is called. A response code is a value representing an application-specific error condition. A response code value of zero is not considered as an error response code. Each response code and response code type is defined in the EDD.

Lexical structure

`abort_on_all_response_codes (VOID)`

The lexical element of the builtin `abort_on_all_response_codes` is shown in Table 54.

Table 54 – Builtin `abort_on_all_response_codes`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
abort_on_all_response_codes();
```

6.3.6 Builtin `ABORT_ON_COMM_ERROR`**Communication profiles: HART**

Purpose: Abort if any communications error is found by the master or the slave

The builtin `ABORT_ON_COMM_ERROR` will set the comm error abort mask in such a way that the method will be aborted if a comm error is found in the device or in the EDD application while sending a command.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`ABORT_ON_COMM_ERROR (VOID)`

The lexical element of the builtin `ABORT_ON_COMM_ERROR` is shown in Table 55.

Table 55 – Builtin `ABORT_ON_COMM_ERROR`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.7 Builtin `abort_on_comm_error`**Communication profiles: FF, ISA100**

Purpose: Sets the mask used for aborting the method when a communications error is detected by the slave device

The builtin `abort_on_comm_error` causes the method to abort if it receives a specific communications error.

When a method is started, it does not abort on a specific communications error until builtin `abort_on_comm_error` is called.

NOTE A communications error of zero is not considered a communications error.

This builtin permits the method writer to select which communications errors should be handled by the method and it frees a method from having to handle a specific communications error code.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary.

Lexical structure

```
abort_on_comm_error(error)
```

The lexical elements of the builtin `abort_on_comm_error` are shown in Table 56.

Table 56 – Builtin `abort_on_comm_error`

Parameter name	Type	Direction	Usage	Description
error	unsigned long	I	m	A communications error code.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = abort_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.8 Builtin `ABORT_ON_COMM_STATUS`

Communication profiles: HART

Purpose: Sets the mask used for aborting the method when a communications error is detected by the slave device

The builtin `ABORT_ON_COMM_STATUS` will set the correct bit(s) in the comm status abort mask in such a way that the specified `comm_status` value will cause the method to abort. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set. The value of `comm_status` may be set to any value from 0x00 to 0x7F.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

```
ABORT_ON_COMM_STATUS(comm_status)
```

The lexical elements of the builtin `ABORT_ON_COMM_STATUS` are shown in Table 57.

Table 57 – Builtin `ABORT_ON_COMM_STATUS`

Parameter name	Type	Direction	Usage	Description
comm_status	unsigned char	I	m	A bit mask for the communication status.
<return>	VOID	—	—	—

6.3.9 Builtin ABORT_ON_DEVICE_STATUS

Communication profiles: HART

Purpose: Set the device status abort mask

The builtin ABORT_ON_DEVICE_STATUS will set the correct bit(s) in the device status abort mask such that the specified device status value will cause the method to abort. The first two bytes of every slave response contain status information. The first byte returned indicates response code status when the most significant bit (bit 7) is 0. The second byte returned, device status, is only meaningful if response code status is the first byte returned. Device status is defined to be the second data octet returned in a transaction. The value of device_status can be set to cause the current method to abort on any combination of device status bits. This is accomplished by creating a mask that will be ANDed to the device status. The value of device_status may be set to any value from 0x00 to 0x7F.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

ABORT_ON_DEVICE_STATUS(device_status)

The lexical elements of the builtin ABORT_ON_DEVICE_STATUS are shown in Table 58.

Table 58 – Builtin ABORT_ON_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
device_status	unsigned int	i	m	A bit mask for the device status.
<return>	VOID	—	—	—

6.3.10 Builtin ABORT_ON_NO_DEVICE (deprecated)

Communication profiles:

Purpose: Abort if no device is found

The builtin ABORT_ON_NO_DEVICE will set the no device abort mask in such a way that the method will be aborted if no device is found while sending a command.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

ABORT_ON_NO_DEVICE(VOID)

The lexical element of the builtin ABORT_ON_NO_DEVICE is shown in Table 59.

Table 59 – Builtin ABORT_ON_NO_DEVICE

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.11 Builtin ABORT_ON_RESPONSE_CODE**Communication profiles: HART****Purpose: Set response code abort mask**

The builtin ABORT_ON_RESPONSE_CODE will set the correct bit in the response code abort mask such that the specified response code value will cause the method to abort. The first two bytes of every slave response contains status information. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0. The response_code value can be set to cause the method to abort on any one response code. The value of response_code may be set to any value from 0 to 127.

To cause the current HART command to be aborted on multiple different response codes, this builtin has to be called once for each response code to be aborted.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method.

The list of all the available masks with default values is shown in Table 60:

Table 60 – Available abort and retry masks

Mask	Default value
response_code_abort_mask	ACCESS_RESTRICTED CMD_NOT_IMPLEMENTED
response_code_retry_mask	All bits set to 0
cmd_48_response_code_abort_mask	ACCESS_RESTRICTED CMD_NOT_IMPLEMENTED
cmd_48_response_code_retry_mask	All bits set to 0
device_status_abort_mask	DEVICE_MALFUNCTION
device_status_retry_mask	All bits set to 0
cmd_48_device_status_abort_mask	DEVICE_MALFUNCTION
cmd_48_device_status_retry_mask	All bits set to 0
comm_status_abort_mask	All bits set to 0
comm_status_retry_mask	All bits set to 1
cmd_48_comm_status_abort_mask	All bits set to 0
cmd_48_comm_status_retry_mask	All bits set to 1
no_device_abort_mask	All bits set to 1
comm_error_abort_mask	All bits set to 1
comm_error_retry_mask	All bits set to 0
cmd_48_no_device_abort_mask	All bits set to 1
cmd_48_comm_error_abort_mask	All bits set to 1
cmd_48_comm_error_retry_mask	All bits set to 0
cmd_48_data_abort_mask	All bits set to 0
cmd_48_data_retry_mask	All bits set to 0

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`ABORT_ON_RESPONSE_CODE(response_code)`

The lexical elements of the builtin `ABORT_ON_RESPONSE_CODE` are shown in Table 61.

Table 61 – Builtin `ABORT_ON_RESPONSE_CODE`

Parameter name	Type	Direction	Usage	Description
<code>response_code</code>	<code>int</code>	<code>I</code>	<code>m</code>	The response code that should be added to the abort mask.
<code><return></code>	<code>VOID</code>	<code>—</code>	<code>—</code>	<code>—</code>

6.3.12 Builtin `abort_on_response_code`

Communication profiles: FF, ISA100

Purpose: Set response code abort mask

The builtin `abort_on_response_code` causes the method to abort upon receiving the specified non-zero response code. This builtin frees a method from having to handle a specific non-zero response code.

When a method is started, it will not abort on a specific response code until this builtin is called. A response code is a value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code type is defined in the EDD.

Lexical structure

`abort_on_response_code(code)`

The lexical elements of the builtin `abort_on_response_code` are shown in Table 62.

Table 62 – Builtin `abort_on_response_code`

Parameter name	Type	Direction	Usage	Description
<code>code</code>	<code>unsigned long</code>	<code>I</code>	<code>m</code>	A bit mask for the response code.
<code><return></code>	<code>long</code>	<code>O</code>	<code>m</code>	One of the return codes specified in Table 376.

Example

```
long status;
status = abort_on_response_code(RC_INVALID_ENTRY);
```

6.3.13 Builtin `ext_send_command`

Communication profiles: HART

Purpose: Send command

The builtin `ext_send_command` has the same functionality as the `send_command` function, except that the command status and data octets returned from the device with command 48 are also returned.

The calling function is responsible for allocating the arrays for the data to be returned. The `cmd_status`, and `more_data_status` VARIABLES are pointers to 3-octet arrays, and the `more_data_info` VARIABLE is a pointer to an array whose size is equal to the number of data octets that can be returned by the device in command 48 (maximum 25). If the status bit indicating that more data is available is returned (status class MORE), command 48 is automatically issued. If command 48 was not issued, the `more_data_status`, and `more_data_info` octets are set to zero.

The octets in the `cmd_status` array and the status bytes from command 48 (if it was sent) are logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried (either the specified command, or command 48). This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. When command 48 is indirectly sent as a result of the device status response, the `cmd 48 xxx` mask values that are set using the `XMTR_xxx` builtins are used to control the aborting and retrying for command 48. The other mask values and related builtins are used to control the aborting and retrying for the directly sent command (including command 48 if it was directly sent).

Lexical structure

`ext_send_command(cmd_number, cmd_status, more_data_status, more_data_info)`

The lexical elements of the builtin `ext_send_command` are shown in Table 63.

Table 63 – Builtin `ext_send_command`

Parameter name	Type	Direction	Usage	Description
<code>cmd_number</code>	int	I	m	The COMMAND number.
<code>cmd_status</code>	char[]	I	m	The COMMAND status.
<code>more_data_status</code>	char[]	I	m	Specifies if more status data is available.
<code>more_data_info</code>	char[]	I	m	Specifies if more info is available.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. Returns <code>BI_SUCCESS</code> if the command was successful, <code>BI_COMM_ERROR</code> if an error occurred sending the command, or <code>BI_NO_DEVICE</code> if no device was found.

6.3.14 Builtin `ext_send_command_trans`

Communication profiles: HART

Purpose: Send specified command transaction

The builtin `ext_send_command_trans` sends the command with the specified transaction to the device. This function is to be used to send commands that have been defined with multiple transactions.

builtin `ext_send_command_trans` has the same functionality as the `send_command_trans` function, except that the command status and data octets returned from the device with command 48 are also returned.

The calling function is responsible for allocating the arrays for the data to be returned. The `cmd_status`, and `more data status` variables are pointers to 3-octet arrays, and the `more-data-info` variable is a pointer to an array whose size is equal to the number of data octets that can be returned by the device in command 48 (maximum 25). If command 48 was not issued, the `more_data_status`, and `more_data_info` octets are set to zero.

The octets in the `cmd_status` array and the status bytes from command 48 (if it was sent) are logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried (either the specified command, or command 48). This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. When command 48 is indirectly sent as a result of the device status response, the `cmd 48 xxx` mask values that are set using the `XMTR_xxx` builtins are used to control the aborting and retrying for command 48. The other mask values and related builtins are used to control the aborting and retrying for the directly sent command (including command 48 if it was directly sent).

Lexical structure

```
ext_send_command_trans(cmd_number, transaction, cmd_status,
more_data_status, more_data_info)
```

The lexical elements of the builtin `ext_send_command_trans` are shown in Table 64.

Table 64 – Builtin `ext_send_command_trans`

Parameter name	Type	Direction	Usage	Description
<code>cmd_number</code>	int	I	m	The COMMAND number.
<code>transaction</code>	int	I	m	The number of the TRANSACTION.
<code>cmd_status</code>	char[]	I	m	The COMMAND status.
<code>more_data_status</code>	char[]	I	m	Specifies if more status data is available.
<code>more_data_info</code>	char[]	I	m	Specifies if more info is available.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. Returns <code>BI_SUCCESS</code> if the command was successful, <code>BI_COMM_ERROR</code> if an error occurred sending the command, or <code>BI_NO_DEVICE</code> if no device was found.

6.3.15 Builtin `fail_on_all_comm_errors`

Communication profiles: FF, ISA100

Purpose: Return control to method in case of communication errors

The builtin `fail_on_all_comm_errors` causes control to be returned to the calling method after any communications error is detected by the called builtin instead of aborting the method or retrying the communications request.

Instead of aborting the method or attempting to retry a failing communications request when any communications error occurs, the builtin `fail_on_all_comm_errors` initializes the builtins to send a communications error code to the method.

When a communications error occurs, the builtin performing a communications request returns the error `BLTIN_FAIL_COMM_ERROR`. The method can call the builtin `get_comm_error` and subsequently `get_comm_error_string` to obtain additional information about the error and later display the error string, if appropriate.

Lexical structure

```
fail_on_all_comm_errors(VOID)
```

The lexical element of the builtin `fail_on_all_comm_errors` is shown in Table 65.

Table 65 – Builtin fail_on_all_comm_errors

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
fail_on_all_comm_errors();
```

6.3.16 Builtin fail_on_all_response_codes

Communication profiles: FF, ISA100

Purpose: Return control to method in case of response codes

The builtin fail_on_all_response_codes causes control to be returned to the calling method after any non-zero response code is detected by the called builtin instead of aborting the method or retrying.

When a response code error occurs, any builtin performing a request returns the error BLTIN_FAIL_RESPONSE_CODE. The method can call the builtin get_comm_error and subsequently get_comm_error_string to obtain additional information about the error, if appropriate.

A response code is a value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

Lexical structure

```
fail_on_all_response_codes(VOID)
```

The lexical element of the builtin fail_on_all_response_codes is shown in Table 66.

Table 66 – Builtin fail_on_all_response_codes

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
fail_on_all_response_codes();
```

6.3.17 Builtin fail_on_comm_error

Communication profiles: FF, ISA100

Purpose: Return control to method in case of specified communication error

The builtin fail_on_comm_error causes control to be returned to the calling method after the specified communications error is detected by the called builtin instead of aborting the method or retrying.

When the specified communications error occurs, any builtin performing a communications request returns the error BLTIN_FAIL_COMM_ERROR. The method can call the builtin get_comm_error and then get_comm_error_string to obtain additional information about the error and later display it, if appropriate.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary. A communications error of zero is not considered a communications error.

Lexical structure

```
fail_on_comm_error(error)
```

The lexical elements of the builtin fail_on_comm_error are shown in Table 67.

Table 67 – Builtin fail_on_comm_error

Parameter name	Type	Direction	Usage	Description
error	unsigned long	I	m	A communication error code.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = fail_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.18 Builtin fail_on_response_code

Communication profiles: FF, ISA100

Purpose: Return control to method in case of specified response code

The builtin fail_on_response_code causes control to be returned to the calling method after a specific non-zero response code is detected by the called builtin instead of aborting the method or retrying.

When the specified response code error occurs, any builtin performing a request returns the error BLTIN_FAIL_RESPONSE_CODE. The method can call the builtin get_comm_error and subsequently get_comm_error_string to obtain additional information about the error and later display it, if appropriate.

A response code is a value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

Lexical structure

```
fail_on_response_code(code)
```

The lexical elements of the builtin fail_on_response_code are shown in Table 68.

Table 68 – Builtin fail_on_response_code

Parameter name	Type	Direction	Usage	Description
code	unsigned long	I	m	A response code.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = fail_on_response_code(RC_INVALID_ENTRY);
```

6.3.19 Builtin get_more_status

Communication profiles: HART

Purpose: Get results of command 48

The builtin get_more_status will issue a command 48 to the device and return the response code, communications status, and command status octets in the status array provided. It will also return any data octets returned in the more_data_info array. This function will process the status/data octets in the same manner as the send command.

The calling function is responsible for allocating the arrays for the data to be returned. The parameter more_data_status is a 3-octet array, and the more_data_info is an array whose size is equal to the number of data octets that can be returned by the device in command 48 (maximum 25).

The octets in the more_data_status bytes from command 48 are logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried (either the specified command, or command 48). This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. The cmd 48 xxx mask values that are set using the XMTR_xxx builtins are used to control the aborting and retrying for command 48.

Lexical structure

```
get_more_status(more_data_status, more_data_info)
```

The lexical elements of the builtin get_more_status are shown in Table 69.

Table 69 – Builtin get_more_status

Parameter name	Type	Direction	Usage	Description
more_data_status	char[]		m	Specifies if more info is available.
more_data_info	char[]	l	m	Specifies if more info is available.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the command was successful, BI_COMM_ERROR if an error occurred sending the command, or BI_NO_DEVICE if no device was found.

6.3.20 Builtin IGNORE_ALL_COMM_STATUS

Communication profiles: HART

Purpose: Ignore all communications status bits

The builtin IGNORE_ALL_COMM_STATUS will clear all of the bits in the comm status retry mask and comm status abort mask. This will cause the system to ignore all bits in the comm status value. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the builtin send_command for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`IGNORE_ALL_COMM_STATUS (VOID)`

The lexical element of the builtin `IGNORE_ALL_COMM_STATUS` is shown in Table 70.

Table 70 – Builtin `IGNORE_ALL_COMM_STATUS`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.21 Builtin `IGNORE_ALL_DEVICE_STATUS`

Communication profiles: HART

Purpose: Ignore all device status bits

The builtin `IGNORE_ALL_DEVICE_STATUS` will clear all of the bits in the device status retry mask and device status abort mask. This will cause the system to ignore all bits in the device status octet. The first two bytes of every slave response contains status information. The first byte returned indicates response code status when the most significant bit (bit 7) is 0. The second byte returned, device status, is only meaningful if the first returns response code status. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the builtin `send_command` for implementation of the masks. See `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`IGNORE_ALL_DEVICE_STATUS (VOID)`

The lexical element of the builtin `IGNORE_ALL_DEVICE_STATUS` is shown in Table 71.

Table 71 – Builtin `IGNORE_ALL_DEVICE_STATUS`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.22 Builtin `IGNORE_ALL_RESPONSE_CODES`

Communication profiles: HART

Purpose: Ignore all response codes

The builtin `IGNORE_ALL_RESPONSE_CODES` will clear all of the bits in the response code retry mask and response code abort mask. This will cause the system to ignore all response code values returned from the device.

The first two bytes of every slave response contains status information. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the builtin `send_command` for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`IGNORE_ALL_RESPONSE_CODES (VOID)`

The lexical element of the builtin `IGNORE_ALL_RESPONSE_CODES` is shown in Table 72.

Table 72 – Builtin `IGNORE_ALL_RESPONSE_CODES`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.23 Builtin `IGNORE_COMM_ERROR`

Communication profiles: HART

Purpose: Ignore communication error

The builtin `IGNORE_COMM_ERROR` will set the comm error retry mask and comm error abort mask such that any communications error condition in the device or the EDD application will be ignored while sending a command. The first two bytes of every slave response contains status information. The first byte returned indicates communication status when the most significant bit (bit 7) is 1.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`IGNORE_COMM_ERROR (VOID)`

The lexical element of the builtin `IGNORE_COMM_ERROR` is shown in Table 73.

Table 73 – Builtin `IGNORE_COMM_ERROR`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.24 Builtin IGNORE_COMM_STATUS

Communication profiles: HART

Purpose: Ignore specific communications status bits

The builtin IGNORE_COMM_STATUS will clear the bit(s) in the comm status retry mask and comm status abort mask such that the specified bits in the comm_status value will be ignored. comm_status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 1. The value of comm_status may be set to any value from 0x00 to 0x7F, where each bit corresponds to a communication status returned from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

IGNORE_COMM_STATUS(comm_status)

The lexical elements of the builtin IGNORE_COMM_STATUS are shown in Table 74.

Table 74 – Builtin IGNORE_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
comm_status	int	I	m	The new retry and abort mask for ignoring.
<return>	VOID	—	—	—

6.3.25 Builtin IGNORE_DEVICE_STATUS

Communication profiles: HART

Purpose: Ignore specific device status bits

The builtin IGNORE_DEVICE_STATUS will clear the correct bit(s) in the device status retry mask and device status abort mask such that the specified bits in the device status octet are ignored. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

IGNORE_DEVICE_STATUS(device_status)

The lexical elements of the builtin IGNORE_DEVICE_STATUS are shown in Table 75.

Table 75 – Builtin IGNORE_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
device_status	int	l	m	The new retry and abort mask for ignoring.
<return>	VOID	—	—	—

6.3.26 Builtin IGNORE_NO_DEVICE (deprecated)**Communication profiles: HART****Purpose: Ignore if no device is found**

The builtin IGNORE_NO_DEVICE will clear the correct bits in the no device retry mask and no device abort mask such that the no-device condition should be ignored while sending a command.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

IGNORE_NO_DEVICE(VOID)

The lexical element of the builtin IGNORE_NO_DEVICE is shown in Table 76.

Table 76 – Builtin IGNORE_NO_DEVICE

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.27 Builtin IGNORE_RESPONSE_CODE**Communication profiles: HART****Purpose: Ignore specific response codes**

The builtin IGNORE_RESPONSE_CODE will clear the correct bit in the response code retry mask and response code abort mask such that the specified response code value will be ignored. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0. The value of response_code may be set to any value from 0 to 127.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

IGNORE_RESPONSE_CODE(response_code)

The lexical elements of the builtin IGNORE_RESPONSE_CODE are shown in Table 77.

Table 77 – Builtin IGNORE_RESPONSE_CODE

Parameter name	Type	Direction	Usage	Description
response_code	int	I	m	The response code that should be added to the ignore mask.
<return>	VOID	—	—	—

6.3.28 Builtin read_value

Communication profiles: FF, ISA100

Purpose: Updates a parameter value in the cache by reading it from the device

The builtin read_value causes the value of the parameter to be read from the device and stores the value in the cache.

The builtin read_value does nothing in a non-caching EDD application because a non-caching EDD application does not have a cache to update.

The read_value builtin is subject to the same read and write timeouts as when an application is reading or writing a value.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the variable in question.

Lexical structure

```
read_value(id, member_id)
```

The lexical elements of the builtin read_value are shown in Table 78.

Table 78 – Builtin read_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = read_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(UNIT_CD));
```

6.3.29 Builtin read_value2

Communication profiles: FF, ISA100

Purpose: Reads a variable, record member, or array element from the device

The builtin read_value2 causes the parameter value to be read from the device and stores the value in the cache.

The builtin `read_value2` does nothing in a non-caching EDD application because a non-caching EDD application does not have a cache to update.

The `id` parameter specifies the item id of the block to be accessed, while the `blk_num` parameter specifies the occurrence number of the block to be accessed.

If the `id` parameter references a simple variable, the `member_id` parameter is ignored. If the `id` parameter references a record, the `member_id` parameter specifies a member of the record. If the `id` parameter references an array parameter, the `member_id` parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `blk_num` parameter is specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

If the `id` parameter references a record and the `member_id` parameter equals zero, the entire record is read from the device. Similarly, if the `id` parameter references an array and the `member_id` parameter equals zero the entire array is read from the device.

Lexical structure

```
read_value2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin `read_value2` are shown in Table 79.

Table 79 – Builtin `read_value2`

Parameter name	Type	Direction	Usage	Description
<code>blk_id</code>	unsigned long	I	m	The item id of the BLOCK_A instance to access.
<code>blk_num</code>	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
<code>id</code>	unsigned long	I	o	The item id of the VARIABLE to access.
<code>member_id</code>	unsigned long	I	o	The member id of the VARIABLE to access.
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = read_value2(ITEM_ID(__analog_input_block), 0, MEMBER_ID(PV_SCALE),
MEMBER_ID(UNIT_CD));
```

6.3.30 Builtin ReadCommand

Communication profiles: CS, GPE, PB, PN

Purpose: Execute read command

The builtin `ReadCommand` allows reading variables with a given command.

Lexical structure

```
ReadCommand(command_identifier)
```

The lexical elements of the builtin `ReadCommand` are shown in Table 80.

Table 80 – Builtin ReadCommand

Parameter name	Type	Direction	Usage	Description
command_identifier	reference	I	m	The id of the command to execute.
<return>	long	O	m	A protocol specific error value, the encoding of which is defined by the specific protocol or device.

6.3.31 Builtin retry_on_all_comm_errors**Communication profiles: FF, ISA100****Purpose: Retry builtin in case of communication errors**

The builtin `retry_on_all_comm_errors` causes the last request made by a method to be tried again after any communications error. A request will be retried three times before it fails and an error return is returned.

This builtin frees a method from having to explicitly retry a request after any communications errors.

The communication error codes are defined in the standard profiles and their related strings are defined in the text dictionary.

NOTE A communications error of zero is not considered a communications error.

Lexical structure

```
retry_on_all_comm_errors(VOID)
```

The lexical element of the builtin `retry_on_all_comm_errors` is shown in Table 81.

Table 81 – Builtin retry_on_all_comm_errors

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
retry_on_all_comm_errors();
```

6.3.32 Builtin RETRY_ON_ALL_COMM_STATUS**Communication profiles: HART****Purpose: Set to retry for all comm status**

The builtin `RETRY_ON_ALL_COMM_STATUS` will set all of the bits in the comm status retry mask. This will cause the system to retry the current command if the device returns any comm status value. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks, and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_ALL_COMM_STATUS (VOID)`

The lexical element of the builtin `RETRY_ON_ALL_COMM_STATUS` is shown in Table 82.

Table 82 – Builtin `RETRY_ON_ALL_COMM_STATUS`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.33 Builtin `RETRY_ON_ALL_DEVICE_STATUS`

Communication profiles: HART

Purpose: Set to retry for all device status

The builtin `RETRY_ON_ALL_DEVICE_STATUS` will set all of the bits in the device status retry mask. This will cause the system to retry the current command if the device returns any device status value. The first two bytes of every slave response contains status information. The first byte returned indicates response code status when the most significant bit (bit 7) is 0. The second byte returned, device status, is only meaningful if the first returns response code status. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks, and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_ALL_DEVICE_STATUS (VOID)`

The lexical element of the builtin `RETRY_ON_ALL_DEVICE_STATUS` is shown in Table 83.

Table 83 – Builtin `RETRY_ON_ALL_DEVICE_STATUS`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.34 Builtin `RETRY_ON_ALL_RESPONSE_CODES`

Communication profiles: HART

Purpose: Set to retry for all response codes

The builtin `RETRY_ON_ALL_RESPONSE_CODES` will set all of the bits in the response code retry mask. This will cause the system to retry the current command if the device returns any non-zero response code value.

The first two bytes of every slave response contains status information. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks, and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_ALL_RESPONSE_CODES (VOID)`

The lexical element of the builtin `RETRY_ON_ALL_RESPONSE_CODES` is shown in Table 84.

Table 84 – Builtin `RETRY_ON_ALL_RESPONSE_CODES`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.35 Builtin `retry_on_all_response_codes`

Communication profiles: FF, ISA100

Purpose: Set to retry for all response codes

The builtin `retry_on_all_response_codes` causes the last request made by a method to be retried after any response code. A request will be retried three times before it fails and an error return is returned. This is the default action by the builtins when a method is just started.

This builtin frees a method from having to explicitly retry a request after any response code.

A response code is an integer value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

Lexical structure

`retry_on_all_response_codes (void)`

The lexical element of the builtin `retry_on_all_response_codes` is shown in Table 85.

Table 85 – Builtin `retry_on_all_response_codes`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
retry_on_all_response_codes();
```

6.3.36 Builtin RETRY_ON_COMM_ERROR

Communication profiles: HART

Purpose: Retry if communications error is found

The builtin RETRY_ON_COMM_ERROR will set the comm error retry mask such that the current command will be retried if a comm error is detected in the device or EDD application while sending the command. The first two bytes of every slave response contains status information.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

RETRY_ON_COMM_ERROR(VOID)

The lexical element of the builtin RETRY_ON_COMM_ERROR is shown in Table 86.

Table 86 – Builtin RETRY_ON_COMM_ERROR

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.37 Builtin retry_on_comm_error

Communication profiles: FF, ISA100

Purpose: Retry if communications error is found

The builtin retry_on_comm_error causes the last request made by a method to be retried after a specified communications error. A request will be retried three times before it fails and an error return is returned.

This builtin frees a method from having to explicitly retry a request after a given communications error is received.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary. A communications error of zero is not considered a communications error.

Lexical structure

retry_on_comm_error(error)

The lexical elements of the builtin retry_on_comm_error are shown in Table 87.

Table 87 – Builtin `retry_on_comm_error`

Parameter name	Type	Direction	Usage	Description
error	unsigned long	I	m	The communication errors.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = retry_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.38 Builtin `RETRY_ON_COMM_STATUS`**Communication profiles: HART****Purpose: Set the communications status retry mask**

The builtin `RETRY_ON_COMM_STATUS` will set the correct bit(s) in the comm status retry mask such that any matching set bit of the communication status will cause the current command to be retried. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 1.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_method` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

```
RETRY_ON_COMM_STATUS(comm_status)
```

The lexical elements of the builtin `RETRY_ON_COMM_STATUS` are shown in Table 88.

Table 88 – Builtin `RETRY_ON_COMM_STATUS`

Parameter name	Type	Direction	Usage	Description
comm_status	int	I	m	The new communication status retry mask.
<return>	VOID	—	—	—

6.3.39 Builtin `RETRY_ON_DEVICE_STATUS`**Communication profiles: HART****Purpose: Set device status retry mask**

The builtin `RETRY_ON_DEVICE_STATUS` will set the correct bit(s) in the device status retry mask such that any matching set bit of the device status will cause the current command to be retried. The first two bytes of every slave response contains status information. The first byte returned indicates response code status when the most significant bit (bit 7) is 0. The second byte returned, device status, is only meaningful if the first returns response code status. Device status is defined to be the second data octet returned in a transaction. The value of `device_status` can be set to cause the system to retry the current HART command on any

combination of device status bits. The value of `device_status` may be set to any value from 0x00 to 0x7F, where each bit corresponds to a device status returned from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_DEVICE_STATUS(device_status)`

The lexical elements of the builtin `RETRY_ON_DEVICE_STATUS` are shown in Table 89.

Table 89 – Builtin `RETRY_ON_DEVICE_STATUS`

Parameter name	Type	Direction	Usage	Description
<code>device_status</code>	int	I	m	The new communication status retry mask.
<return>	VOID	—	—	—

6.3.40 Builtin `RETRY_ON_NO_DEVICE` (deprecated)

Communication profiles: CS, GPE, HART

Purpose: Retry if no device is found

The builtin `RETRY_ON_NO_DEVICE` will set the no device retry mask such that the current command will be retried if no device is found while sending a command.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_NO_DEVICE(VOID)`

The lexical element of the builtin `RETRY_ON_NO_DEVICE` is shown in Table 90.

Table 90 – Builtin `RETRY_ON_NO_DEVICE`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.41 Builtin `RETRY_ON_RESPONSE_CODE`

Communication profiles: HART

Purpose: Set response code retry mask

The builtin `RETRY_ON_RESPONSE_CODE` will set the correct bit in the response code retry mask such that the specified response code value will cause the current command to be retried. The first two bytes of every slave response contains status information. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0. The value of `response_code` can be set to cause the system to retry the current HART command on any one response code. The value of `response_code` may be set to any value from 0 to 127.

To cause the current HART command to be retried on multiple different response codes, this builtin has to be called once for each response code to be retried.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of available masks and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`RETRY_ON_RESPONSE_CODE(response_code)`

The lexical elements of the builtin `RETRY_ON_RESPONSE_CODE` are shown in Table 91.

Table 91 – Builtin `RETRY_ON_RESPONSE_CODE`

Parameter name	Type	Direction	Usage	Description
<code>response_code</code>	int	—	m	The response code that should be added to the retry mask.
<return>	VOID	—	—	—

6.3.42 Builtin `retry_on_response_code`

Communication profiles: FF, ISA100

Purpose: Set response code retry mask

The builtin `retry_on_response_code` causes the last request made by a method to be retried three times after a specific response code is received.

This builtin frees a method from having to explicitly retry a request after a given response code is received.

A response code is a value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

Retries the method's last request upon receiving the specified response code.

Lexical structure

`retry_on_response_code(code)`

The lexical elements of the builtin `retry_on_response_code` are shown in Table 92.

Table 92 – Builtin `retry_on_response_code`

Parameter name	Type	Direction	Usage	Description
<code>code</code>	unsigned long	I	m	The response code being resolved.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = retry_on_response_code(RC_INVALID_ENTRY);
```

6.3.43 Builtin send

Communication profiles: HART

Purpose: Send the specified command

The builtin `send` will send the specified command and return the response code, communications status, and command status respectively in the `cmd_status` array. The Device Status byte in the Field Device response shall be inspected. Any bit having Status Class of "MORE" that is Set (1) in Device Status shall result in Command 48 being automatically issued.

The octets in the `cmd_status` array and the status bytes from command 48 (if it was sent) are logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried (either the specified command, or command 48). This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. When command 48 is indirectly sent as a result of the device status response, the `cmd 48 xxx` mask values that are set using the `XMTR_xxx` builtins are used to control the aborting and retrying for command 48. The other mask values and related builtins are used to control the aborting and retrying for the directly sent command (including command 48 if it was directly sent).

The status octets for the command are returned in the `cmd_status` parameter, a pointer to a 3-octet array of unsigned char, which is allocated in the method making the call.

Lexical structure

```
send(cmd_number, cmd_status)
```

The lexical elements of the builtin `send` are shown in Table 93.

Table 93 – Builtin `send`

Parameter name	Type	Direction	Usage	Description
<code>cmd_number</code>	int	I	m	The number of command to send.
<code>cmd_status</code>	char*	O	m	The status octets of the send command.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the command was successful, BI_COMM_ERROR if an error occurred sending the command, or BI_NO_DEVICE if no device was found.

Example

```
int result;
char cmd_status[STATUS_SIZE];
result = send(40, cmd_status);
```

In the above example, if any bits of Status Class "MORE" are set in cmd_status[STATUS_DEVICE_STATUS] is set, Command 48 will automatically be dispatched before the send function returns.

6.3.44 Builtin send_all_values

Communication profiles: FF, ISA100

Purpose: send modified values to device

The builtin send_all_values sends all of the values in the variable table that the method modified but has not previously sent to the device. This action changes the device values to match the values set by the method.

This builtin does nothing for a non-caching EDD application because it does not have a variable table.

Lexical structure

```
send_all_values(VOID)
```

The lexical element of the builtin send_all_values is shown in Table 94.

Table 94 – Builtin send_all_values

Parameter name	Type	Direction	Usage	Description
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = send_all_values();
```

6.3.45 Builtin send_command

Communication profiles: HART

Purpose: Send the specified command

The builtin send_command will send the specified command. The status/data octets from the command are then logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried. This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. No status or data octets are returned with this function. The cmd 48 xxx mask values that are set using the XMTR_xxx builtins do not affect the aborting or retrying for the send_command builtin.

Lexical structure

```
send_command(cmd_number)
```

The lexical elements of the builtin send_command are shown in Table 95.

Table 95 – Builtin send_command

Parameter name	Type	Direction	Usage	Description
cmd_number	int	I	m	The number of command to send.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the function was successful, BI_COMM_ERROR if an error occurred sending the HART command, or BI_NO_DEVICE if no device was found.

6.3.46 Builtin send_command_trans

Communication profiles: HART

Purpose: Send the command with the specified transaction

The builtin send_command_trans sends the command with the specified transaction to the device. This function is to be used to send commands that have been defined with multiple transactions.

The builtin send_command_trans will send the specified command. The status/data octets from the command are then logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried. This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. No status or data octets are returned with this command. The cmd 48 xxx mask values that are set using the XMTR_xxx builtins do not affect the aborting or retrying for the send_command builtin.

Lexical structure

```
send_command_trans(cmd_number, transaction)
```

The lexical elements of the builtin send_command_trans are shown in Table 96.

Table 96 – Builtin send_command_trans

Parameter name	Type	Direction	Usage	Description
cmd_number	int	I	m	The number of command to send.
transaction	int	I	m	The number of the TRANSACTION.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the function was successful, BI_COMM_ERROR if an error occurred sending the HART command, or BI_NO_DEVICE if no device was found.

6.3.47 Builtin send_trans

Communication profiles: HART

Purpose: Send the command with the specified transaction

The builtin send_trans sends a command with the specified transaction to the device. This function is to be used to send commands that have been defined with multiple transactions.

The builtin `send_trans` will send the specified command and return the response code, communications status, and device status in the `cmd_status` array. The Device Status byte in the Field Device response shall be inspected. Any bit having Status Class of "MORE" that is Set (1) in Device Status shall result in Command 48 being automatically issued.

The octets in the `cmd_status` array and the status bytes from command 48 (if it was sent) are logically ANDed with the appropriate abort masks to determine if the method should be aborted. If there is a match, the method is aborted and the abort methods (if any were designated) are run. If there is no match for the abort masks, the same data is ANDed with the appropriate retry masks. If there is a match, the command will be retried (either the specified command, or command 48). This continues until either the command succeeds, an abort occurs, or the maximum number of retries is reached. When command 48 is indirectly sent as a result of the device status response, the `cmd 48 xxx` mask values that are set using the `XMTR_xxx` builtins are used to control the aborting and retrying for command 48. The other mask values and related builtins are used to control the aborting and retrying for the directly sent command (including command 48 if it was directly sent).

The status octets for the command are returned in the `cmd_status` parameter, a 3-octet array, allocated in the method making the call.

Lexical structure

```
send_trans(cmd_number, transaction, cmd_status)
```

The lexical elements of the builtin `send_trans` are shown in Table 97.

Table 97 – Builtin `send_trans`

Parameter name	Type	Direction	Usage	Description
<code>cmd_number</code>	int	I	m	The number of command to send.
<code>transaction</code>	int	I	m	The number of the TRANSACTION.
<code>cmd_status</code>	char*	O	m	The status octets of the send command.
<return>	int	O	m	One of the return codes specified in Table 377. <code>BI_SUCCESS</code> if the command was successful, <code>BI_COMM_ERROR</code> if an error occurred sending the command, or <code>BI_NO_DEVICE</code> if no device was found.

6.3.48 Builtin `send_value`

Communication profiles: FF, ISA100

Purpose: Sends a variable table value to the device

The builtin `send_value` causes the value of the specified variable in the cache to be sent to the device, thus updating the device value. This builtin does nothing in a non-caching EDD application, because a non-caching EDD application does not have a cache.

Some values are `READ_ONLY` in the device and cannot be written. Attempting to do so will cause a builtin error to be returned.

The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the variable in question.

Lexical structure

```
send_value(id, member_id)
```

The lexical elements of the builtin `send_value` are shown in Table 98.

Table 98 – Builtin send_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to read.
member_id	unsigned long	I	m	The member id of the VARIABLE to read.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long id;
id = ITEM_ID(PARAM.HI_LIM);
status = edit_device_value ("|en|Enter new HI_LIM value:\n", 0, 0, 0, id,
0); status = send_value(id, 0);

```

6.3.49 Builtin send_value2**Communication profiles: FF, ISA100**

Purpose: Sends the value of a variable, record member, or array element to the device

The builtin send_value2 causes the value of the specified variable, record member, or array element in the cache to be sent to the device, thus updating the device value. This builtin does nothing in a non-caching EDD application, because a non-caching EDD application does not have a cache.

Some values are READ_ONLY in the device and cannot be written. Attempting to do so will cause a builtin error to be returned.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple variable, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID. The blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

If the id parameter references a record and the member_id parameter equals zero, the entire record is written to the device.

Similarly, if the id parameter references an array and the member_id parameter equals zero, the entire array is written to the device.

Lexical structure

```
send_value2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin send_value2 are shown in Table 99.

Table 99 – Builtin send_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The item id of the VARIABLE to read.
member_id	unsigned long	I	m	The member id of the VARIABLE to read.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long aiBlock;
unsigned long id;

aiBlock = ITEM_ID(__analog_input_block);
id = MEMBER_ID(HI_LIM);

status = edit_device_value2("|en|Enter new HI_LIM value:\n", 0, 0, 0, 0, 0,
aiBlock, 0, id, 0);
status = send_value2(aiBlock, 0, id, 0);

```

6.3.50 Builtin SET_NUMBER_OF_RETRIES**Communication profiles: CS, GPE, HART****Purpose: Set the number of times a command is retried**

The builtin SET_NUMBER_OF_RETRIES sets up the number of times a command will be retried due to the appropriate retry masks. This value is defaulted to 0 at the beginning of each method.

Lexical structure

```
SET_NUMBER_OF_RETRIES(n)
```

The lexical elements of the builtin SET_NUMBER_OF_RETRIES are shown in Table 100.

Table 100 – Builtin SET_NUMBER_OF_RETRIES

Parameter name	Type	Direction	Usage	Description
n	int	I	m	The number of retries of a command.
<return>	int	O	m	One of the return codes specified in Table 377.

6.3.51 Builtin WriteCommand**Communication profiles: CS, GPE, PB, PN****Purpose: Execute write command**

The builtin WriteCommand allows to write variables with a given command.

Lexical structure

```
WriteCommand(command_identifier)
```

The lexical elements of the builtin WriteCommand are shown in Table 101.

Table 101 – Builtin WriteCommand

Parameter name	Type	Direction	Usage	Description
command_identifier	reference	I	m	The Identifier of the COMMAND to execute.
<return>	long	O	m	A protocol specific error value, the encoding of which is defined by the specific protocol or device.

6.3.52 Builtin XMTR_ABORT_ON_ALL_COMM_STATUS

Communication profiles: HART

Purpose: Set to abort for all command 48 communication status

The builtin XMTR_ABORT_ON_ALL_COMM_STATUS will set all of the bits in the command 48 comm status abort mask. This will cause the system to abort the current method if the device returns any command 48 comm status value. Comm status is defined to be the first octet returned in a transaction, when bit 7 of this octet is set. The comm status can be set to a value from 0x00 to 0x7F, where each bit corresponds to a communications error received from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_ABORT_ON_ALL_COMM_STATUS(VOID)

The lexical element of the builtin XMTR_ABORT_ON_ALL_COMM_STATUS is shown in Table 102.

Table 102 – Builtin XMTR_ABORT_ON_ALL_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.53 Builtin XMTR_ABORT_ON_ALL_DATA

Communication profiles: HART

Purpose: Set to abort for all command 48 data

The builtin XMTR_ABORT_ON_ALL_DATA will set all of the bits in the command 48 data abort masks. This will cause the system to abort if any bit in command 48 is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will be valid only during the current method. See send_command for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks, and their default values.

The mask affected by this function is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY` functions.

Lexical structure

`XMTR_ABORT_ON_ALL_DATA(VOID)`

The lexical element of the builtin `XMTR_ABORT_ON_ALL_DATA` is shown in Table 103.

Table 103 – Builtin XMTR_ABORT_ON_ALL_DATA

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.54 Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS

Communication profiles: HART

Purpose: Set to abort for all command 48 communications status

The builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS` will set all of the bits in the command 48 device status abort mask. This will cause the system to abort the current method if the device returns any command 48 device status value. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`XMTR_ABORT_ON_ALL_DEVICE_STATUS(VOID)`

The lexical element of the builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS` is shown in Table 104.

Table 104 – Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.55 Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES

Communication profiles: HART

Purpose: Set to abort for all command 48 response codes

The builtin `XMTR_ABORT_ON_ALL_RESPONSE_CODES` will set all of the bits in the command 48 response code abort mask. This will cause the system to abort the current method if the device returns any command 48 response code value.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function

for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_ABORT_ON_ALL_RESPONSE_CODES (VOID)

The lexical element of the builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES is shown in Table 105.

Table 105 – Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.56 Builtin XMTR_ABORT_ON_COMM_ERROR

Communication profiles: HART

Purpose: Abort if communications error is found during command 48

The builtin XMTR_ABORT_ON_COMM_ERROR will set the command 48 comm error abort mask such that the method will be aborted if a comm error is found while sending command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_ABORT_ON_COMM_ERROR (VOID)

The lexical element of the builtin XMTR_ABORT_ON_COMM_ERROR is shown in Table 106.

Table 106 – Builtin XMTR_ABORT_ON_COMM_ERROR

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.57 Builtin XMTR_ABORT_ON_COMM_STATUS

Communication profiles: HART

Purpose: Add status abort mask to command 48

The builtin XMTR_ABORT_ON_COMM_STATUS will set the correct bit(s) in the command 48 comm status abort mask such that the specified command 48 comm status value will cause the method to abort. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 1. The comm_status_value can be set to a value from 0 to 127, where each bit corresponds to a communications error received from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`XMTR_ABORT_ON_COMM_STATUS(comm_status)`

The lexical elements of the builtin `XMTR_ABORT_ON_COMM_STATUS` are shown in Table 107.

Table 107 – Builtin `XMTR_ABORT_ON_COMM_STATUS`

Parameter name	Type	Direction	Usage	Description
<code>comm_status</code>	<code>int</code>	<code>l</code>	<code>m</code>	The new comm status abort mask.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.58 Builtin `XMTR_ABORT_ON_DATA`

Communication profiles: HART

Purpose: Abort if specified data bit in command 48 is set

The builtin `XMTR_ABORT_ON_DATA` will set the correct bit(s) in the command 48 abortmask such that the method will be aborted if the specified bit is set in the data field returned by command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`XMTR_ABORT_ON_DATA(data, bit)`

The lexical elements of the builtin `XMTR_ABORT_ON_DATA` are shown in Table 108.

Table 108 – Builtin `XMTR_ABORT_ON_DATA`

Parameter name	Type	Direction	Usage	Description
<code>data</code>	<code>int</code>	<code>l</code>	<code>m</code>	Zero based index into command 48 data field.
<code>bit</code>	<code>int</code>	<code>l</code>	<code>m</code>	Bit number (0..7). Bit 0 is the least significant bit.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.59 Builtin XMTR_ABORT_ON_DEVICE_STATUS

Communication profiles: HART

Purpose: Add status abort mask to command 48

The builtin XMTR_ABORT_ON_DEVICE_STATUS will set the correct bit(s) in the command 48 device status abort mask in such a way that any matching set bit of the command 48 device status will cause the method to abort. Device status is defined to be the second data octet returned in a transaction. The device_status value can be set to a value from 0 to 127, where each bit corresponds to a device status returned from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this function is used by the send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_ABORT_ON_DEVICE_STATUS(device_status)

The lexical elements of the builtin XMTR_ABORT_ON_DEVICE_STATUS are shown in Table 109.

Table 109 – Builtin XMTR_ABORT_ON_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
device_status	int	I	m	The new device status abort mask.
<return>	VOID	—	—	—

6.3.60 Builtin XMTR_ABORT_ON_NO_DEVICE (deprecated)

Communication profiles:

Purpose: Abort if no device is found during command 48

The builtin XMTR_ABORT_ON_NO_DEVICE will set the command 48 no device mask such that the method will be aborted if no device is found while sending a command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_ABORT_ON_NO_DEVICE(VOID)

The lexical element of the builtin XMTR_ABORT_ON_NO_DEVICE is shown in Table 110.

Table 110 – Builtin XMTR_ABORT_ON_NO_DEVICE

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.61 Builtin XMTR_ABORT_ON_RESPONSE_CODE**Communication profiles: HART****Purpose: Add response code abort mask to command 48**

The builtin XMTR_ABORT_ON_RESPONSE_CODE will set the correct bit in the command 48 response code abort mask such that the specified command 48 response code value will cause the method to abort. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

```
XMTR_ABORT_ON_RESPONSE_CODE(response_code)
```

The lexical elements of the builtin XMTR_ABORT_ON_RESPONSE_CODE are shown in Table 111.

Table 111 – Builtin XMTR_ABORT_ON_RESPONSE_CODE

Parameter name	Type	Direction	Usage	Description
response_code	int	l	m	The new response code abort mask.
<return>	VOID	—	—	—

6.3.62 Builtin XMTR_IGNORE_ALL_COMM_STATUS**Communication profiles: HART****Purpose: Ignore all command 48 status bits**

The builtin XMTR_IGNORE_ALL_COMM_STATUS will clear all of the bits in the command 48 comm status retry mask and command 48 comm status abort mask. This will cause the system to ignore all bits in the comm status value. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

XMTR_IGNORE_ALL_COMM_STATUS (VOID)

The lexical element of the builtin XMTR_IGNORE_ALL_COMM_STATUS is shown in Table 112.

Table 112 – Builtin XMTR_IGNORE_ALL_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.63 Builtin XMTR_IGNORE_ALL_DATA

Communication profiles: HART

Purpose: Ignore all command 48 device status bits

The builtin XMTR_IGNORE_ALL_DATA will clear all of the bits in the command 48 data retry mask and command 48 data abort mask. This will cause the system to ignore all data in the command 48 response message.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will be valid only during the current method. See send_command for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks, and their default values.

The mask affected by this function is used by the send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY functions.

Lexical structure

XMTR_IGNORE_ALL_DATA (VOID)

The lexical element of the builtin XMTR_IGNORE_ALL_DATA is shown in Table 113.

Table 113 – Builtin XMTR_IGNORE_ALL_DATA

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.64 Builtin XMTR_IGNORE_ALL_DEVICE_STATUS

Communication profiles: HART

Purpose: Ignore all command 48 device status bits

The builtin XMTR_IGNORE_ALL_DEVICE_STATUS will clear all of the bits in the command 48 device status retry mask and command 48 device status abort mask. This will cause the system to ignore all bits in the command 48 device status octet. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_IGNORE_ALL_DEVICE_STATUS (VOID)

The lexical element of the builtin XMTR_IGNORE_ALL_DEVICE_STATUS is shown in Table 114.

Table 114 – Builtin XMTR_IGNORE_ALL_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.65 Builtin XMTR_IGNORE_ALL_RESPONSE_CODES

Communication profiles: HART

Purpose: Ignore all command 48 response codes

The builtin XMTR_IGNORE_ALL_RESPONSE_CODES will clear all of the bits in the command 48 response code retry mask and command 48 response code abort mask. This will cause the system to ignore all response code values returned from the device.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_IGNORE_ALL_RESPONSE_CODES (VOID)

The lexical element of the builtin XMTR_IGNORE_ALL_RESPONSE_CODES is shown in Table 115.

Table 115 – Builtin XMTR_IGNORE_ALL_RESPONSE_CODES

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.66 Builtin XMTR_IGNORE_COMM_ERROR

Communication profiles: HART

Purpose: Ignore if communications error is found during command 48

The builtin XMTR_IGNORE_COMM_ERROR will clear all bits in the command 48 comm error retry mask and command 48 comm error abort mask such that a communications error condition will be ignored while sending command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_IGNORE_COMM_ERROR(VOID)

The lexical element of the builtin XMTR_IGNORE_COMM_ERROR is shown in Table 116.

Table 116 – Builtin XMTR_IGNORE_COMM_ERROR

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.67 Builtin XMTR_IGNORE_COMM_STATUS

Communication profiles: HART

Purpose: Ignore specific command 48 communications status bits

The builtin XMTR_IGNORE_COMM_STATUS will clear the correct bit(s) in the command 48 comm status retry mask and command 48 comm status abort mask such that the specified bits in the command 48 comm status value will be ignored. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this octet is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_IGNORE_COMM_STATUS(comm_status)

The lexical elements of the builtin XMTR_IGNORE_COMM_STATUS are shown in Table 117.

Table 117 – Builtin XMTR_IGNORE_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
comm_status	int	l	m	The new comm status abort and retry mask.
<return>	VOID	—	—	—

6.3.68 Builtin XMTR_IGNORE_DATA

Communication profiles: HART

Purpose: Ignore specific Command 48 data bits

The builtin XMTR_IGNORE_DATA shall reset the specified bit, in the specified byte, in the command 48 data retry mask and command 48 data abort data mask such that the method will ignore the specified bit in the data field returned by command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will be valid only during the current method. See send_command for implementation of the masks.

See ABORT_ON_RESPONSE_CODE for a list of the available masks, and their default values.

The mask affected by this builtin is used by the builtin `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY` functions.

Lexical structure

```
XMTR_IGNORE_DATA(data, bit)
```

The lexical elements of the builtin `XMTR_IGNORE_DATA` are shown in Table 118.

Table 118 – Builtin XMTR_IGNORE_DATA

Parameter name	Type	Direction	Usage	Description
Data	int	l	m	Zero based index into command 48 data field.
Bit	int	l	m	Bit number (0..7). Bit 0 is the least significant bit.
<return>	VOID	—	—	—

6.3.69 Builtin XMTR_IGNORE_DEVICE_STATUS

Communication profiles: HART

Purpose: Ignore specific command 48 status bits

The builtin `XMTR_IGNORE_DEVICE_STATUS` will clear the correct bit(s) in the command 48 device status retry mask and command 48 device status abort mask such that the specified bits in the command 48 device status octet are ignored. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

```
XMTR_IGNORE_DEVICE_STATUS(device_status)
```

The lexical elements of the builtin `XMTR_IGNORE_DEVICE_STATUS` are shown in Table 119.

Table 119 – Builtin XMTR_IGNORE_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
device_status	int	l	m	The bits to be cleared in the abort and retry mask.
<return>	VOID	—	—	—

6.3.70 Builtin XMTR_IGNORE_NO_DEVICE (deprecated)

Communication profiles: HART

Purpose: Ignore if no device is found during command 48

The builtin `XMTR_IGNORE_NO_DEVICE` will set the command 48 no device retry mask and command 48 no device abort mask such that the no device condition should be ignored while sending a command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`XMTR_IGNORE_NO_DEVICE(VOID)`

The lexical element of the builtin `XMTR_IGNORE_NO_DEVICE` is shown in Table 120.

Table 120 – Builtin `XMTR_IGNORE_NO_DEVICE`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.71 Builtin `XMTR_IGNORE_RESPONSE_CODE`

Communication profiles: HART

Purpose: Ignore specific command 48 response codes

The builtin `XMTR_IGNORE_RESPONSE_CODE` will clear the correct bit in the command 48 response code retry mask and command 48 response code abort mask in such a way that the specified command 48 response code value will be ignored. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the `send_command` function for implementation of the masks. See `ABORT_ON_RESPONSE_CODE` for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY`.

Lexical structure

`XMTR_IGNORE_RESPONSE_CODE(response_code)`

The lexical elements of the builtin `XMTR_IGNORE_RESPONSE_CODE` are shown in Table 121.

Table 121 – Builtin `XMTR_IGNORE_RESPONSE_CODE`

Parameter name	Type	Direction	Usage	Description
response_code	int	l	m	The new response code mask.
<return>	VOID	—	—	—

6.3.72 Builtin XMTR_RETRY_ON_ALL_COMM_STATUS

Communication profiles: HART

Purpose: Set to retry for all command 48 communications status

The builtin XMTR_RETRY_ON_ALL_COMM_STATUS will set all of the bits in the command 48 comm status retry mask. This will cause the system to retry command 48 if the device returns any command 48 communications status value. Comm status is defined to be the first data byte returned in a HART transaction, when bit 7 of this byte is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will be valid only during the current method. See send_command for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY functions.

Lexical structure

XMTR_RETRY_ON_ALL_COMM_STATUS (VOID)

The lexical element of the builtin XMTR_RETRY_ON_ALL_COMM_STATUS is shown in Table 122.

Table 122 – Builtin XMTR_RETRY_ON_ALL_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.73 Builtin XMTR_RETRY_ON_ALL_DATA

Communication profiles: HART

Purpose: Set to retry for all command 48 device status

The builtin XMTR_RETRY_ON_ALL_DATA will set all of the bits in the command 48 data retry mask. This will cause the system to retry if any bit in command 48 is set.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will be valid only during the current method. See send_command for implementation of the masks. See ABORT_ON_RESPONSE_CODE for a list of the available masks, and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY functions.

Lexical structure

XMTR_RETRY_ON_ALL_DATA (VOID)

The lexical element of the builtin XMTR_RETRY_ON_ALL_DATA is shown in Table 123.

Table 123 – Builtin XMTR_RETRY_ON_ALL_DATA

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.74 Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS

Communication profiles: HART

Purpose: Set to retry for all command 48 device status

The builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS will set all of the bits in the command 48 device status retry mask. This will cause the system to retry command 48 if the device returns any command 48 device status value. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_RETRY_ON_ALL_DEVICE_STATUS (VOID)

The lexical element of the builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS is shown in Table 124.

Table 124 – Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.75 Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES

Communication profiles: HART

Purpose: Set to retry for all command 48 response codes

The builtin XMTR_RETRY_ON_ALL_RESPONSE_CODE will set all of the bits in the command 48 response code retry mask. This will cause the system command 48 to be retried if the device returns any command 48 response code value.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_RETRY_ON_ALL_RESPONSE_CODES (VOID)

The lexical element of the builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES is shown in Table 125.

Table 125 – Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.76 Builtin XMTR_RETRY_ON_COMM_ERROR**Communication profiles:** HART**Purpose:** Retry if communications error is found during command 48

The builtin XMTR_RETRY_ON_COMM_ERROR will set the command 48 comm error retry mask such that command 48 will be retried if a comm error is found while sending command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

```
XMTR_RETRY_ON_COMM_ERROR(VOID)
```

The lexical element of the builtin XMTR_RETRY_ON_COMM_ERROR is shown in Table 126.

Table 126 – Builtin XMTR_RETRY_ON_COMM_ERROR

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.77 Builtin XMTR_RETRY_ON_COMM_STATUS**Communication profiles:** HART**Purpose:** Add status retry mask to command 48

The builtin XMTR_RETRY_ON_COMM_STATUS will set the correct bit(s) in the command 48 comm status retry mask in such a way that the specified command 48 comm status value will cause command 48 to be retried. Comm status is defined to be the first data octet returned in a transaction, when bit 7 of this is 1.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

```
XMTR_RETRY_ON_COMM_STATUS(comm_status)
```

The lexical elements of the builtin XMTR_RETRY_ON_COMM_STATUS are shown in Table 127.

Table 127 – Builtin XMTR_RETRY_ON_COMM_STATUS

Parameter name	Type	Direction	Usage	Description
comm_status	int	l	m	The new comm status retry mask.
<return>	VOID	—	—	—

6.3.78 Builtin XMTR_RETRY_ON_DATA**Communication profiles: HART****Purpose: Retry if specified data bit in command 48 is set**

The builtin XMTR_RETRY_ON_DATA will set the specified bit(s) in the command 48 data retry mask in such a way that the command 48 will be retried if the specified bit is set in the data field returned by command 48.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

```
XMTR_RETRY_ON_DATA(data, bit)
```

The lexical elements of the builtin XMTR_RETRY_ON_DATA are shown in Table 128.

Table 128 – Builtin XMTR_RETRY_ON_DATA

Parameter name	Type	Direction	Usage	Description
data	int	l	m	Zero based index into command 48 data field.
bit	int	l	m	Bit number (0..7). Bit 0 is the least significant bit.
<return>	VOID	—	—	—

6.3.79 Builtin XMTR_RETRY_ON_DEVICE_STATUS**Communication profiles: HART****Purpose: Add status retry mask to command 48**

The builtin XMTR_RETRY_ON_DEVICE_STATUS will set the correct bit(s) in the command 48 device status retry mask in such a way that any matching set bit of the command 48 device status will cause command 48 to be retried. Device status is defined to be the second data octet returned in a transaction.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_RETRY_ON_DEVICE_STATUS(device_status)

The lexical elements of the builtin XMTR_RETRY_ON_DEVICE_STATUS are shown in Table 129.

Table 129 – Builtin XMTR_RETRY_ON_DEVICE_STATUS

Parameter name	Type	Direction	Usage	Description
device_status	INT4	l	m	The new device status retry mask.
<return>	VOID	—	—	—

6.3.80 Builtin XMTR_RETRY_ON_NO_DEVICE (deprecated)

Communication profiles: HART

Purpose: Retry if no device is found during command 48

The builtin XMTR_RETRY_ON_NO_DEVICE will set the command 48 nodevice retry mask in such a way that command 48 will be retried if no device is found while sending command 48.

The retry and abort masks are reset to their values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_RETRY_ON_NO_DEVICE(VOID)

The lexical element of the builtin XMTR_RETRY_ON_NO_DEVICE is shown in Table 130.

Table 130 – Builtin XMTR_RETRY_ON_NO_DEVICE

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.3.81 Builtin XMTR_RETRY_ON_RESPONSE_CODE

Communication profiles: HART

Purpose: Add response code retry mask to command 48

The builtin XMTR_RETRY_ON_RESPONSE_CODE will set the correct bit in command 48 the response code retry mask in such a way that the specified command 48 response code value will cause command 48 to be retried. The response code is defined to be the first data octet returned in a transaction, when bit 7 of this octet is 0.

The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See the send_command function for implementation of the masks. See builtin ABORT_ON_RESPONSE_CODE for a list of the available masks and their default values.

The mask affected by this builtin is used by the builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY.

Lexical structure

XMTR_RETRY_ON_RESPONSE_CODE(response_code)

The lexical element of the builtin XMTR_RETRY_ON_RESPONSE_CODE is shown in Table 131.

Table 131 – Builtin XMTR_RETRY_ON_RESPONSE_CODE

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.4 Action builtins

6.4.1 Builtin fgetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get VARIABLE value

The builtin fgetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

fgetval(VOID)

The lexical element of the builtin fgetval is shown in Table 132.

Table 132 – Builtin fgetval

Parameter name	Type	Direction	Usage	Description
<return>	double	O	m	The value of the VARIABLE.

Example

```
double fval;
fval = fgetval();
fval = fval + 10;
```

6.4.2 Builtin fsetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Set VARIABLE value

The builtin fsetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

`fsetval(value)`

The lexical elements of the builtin `fsetval` are shown in Table 133.

Table 133 – Builtin `fsetval`

Parameter name	Type	Direction	Usage	Description
value	double	I	m	The new value for the VARIABLE.
<return>	double	O	m	The value of the VARIABLE.

6.4.3 Builtin `get_date`

Communication profiles: FF, ISA100

Purpose: Get current value of a date VARIABLE

The builtin `get_date` is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The methods associated with pre-edit, post-edit, post-read, or pre-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read or written by an application, a user method, or an edit method.

Lexical structure

`get_date(data, size)`

The lexical elements of the builtin `get_date` are shown in Table 134.

Table 134 – Builtin `get_date`

Parameter name	Type	Direction	Usage	Description
data	char[]	O	m	The returned value of the VARIABLE. The structure of this octet buffer is defined in Annex B of IEC 61804-3:___ ⁶ .
size	long	I/O	m	The returned length of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status, date_size;
char date_value[8];

date_size = 8;
status = get_date(date_value, &date_size);
```

⁶ Under preparation. Stage at the time of publication: IEC RPUB 61804-4:2020.

6.4.4 Builtin get_double

Communication profiles: FF, ISA100

Purpose: Get double value from VARIABLE

The builtin get_double is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of the VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The methods associated with pre-edit, post-edit, post-read, or pre-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read by an application or another method.

Lexical structure

```
get_double(value)
```

The lexical elements of the builtin get_double are shown in Table 135.

Table 135 – Builtin get_double

Parameter name	Type	Direction	Usage	Description
value	double	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
double value;
status = get_double(&value);
```

6.4.5 Builtin get_float

Communication profiles: FF, ISA100

Purpose: Get float value from VARIABLE

The builtin get_float is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from a connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The methods associated with pre-edit, post-edit, post-read, or pre-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read by an application, a user method, or an edit method.

Lexical structure

```
get_float(value)
```

The lexical elements of the builtin `get_float` are shown in Table 136.

Table 136 – Builtin `get_float`

Parameter name	Type	Direction	Usage	Description
value	float	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
float value;
status = get_float(&value);
```

6.4.6 Builtin `get_signed`

Communication profiles: FF, ISA100

Purpose: Get scaled signed value of VARIABLE

The builtin `get_signed` is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The methods associated with pre-edit, post-edit, post-read, or pre-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read by an application, a user method, or an edit method.

Lexical structure

```
get_signed(value)
```

The lexical elements of the builtin `get_signed` are shown in Table 137.

Table 137 – Builtin `get_signed`

Parameter name	Type	Direction	Usage	Description
value	long	O	m	The returned value of the signed VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
long value;
status = get_signed(&value);
```

6.4.7 Builtin get_string

Communication profiles: FF, ISA100

Purpose: Get string value of VARIABLE

The builtin get_string is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from a connected device. Custom calculations may then be performed on the VARIABLE value before it is saved in the EDD application.

VARIABLEs of these data types shall be handled by the method because they are not supported by the programming language-C. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The method associated with pre-edit, post-edit, post-read, or post-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read by an application, a user method, or an edit method.

Lexical structure

```
get_string(string, len)
```

The lexical elements of the builtin get_string are shown in Table 138.

Table 138 – Builtin get_string

Parameter name	Type	Direction	Usage	Description
string	char*	O	m	The buffer for the returned string.
len	long*	I/O	m	The maximum size of the string. The returned length of the string.
<return>	int	O	m	One of the return codes specified in Table 376.

Example

```
long status;
long len;
char buf[256];
status = get_string(buf, &len);
```

6.4.8 Builtin get_unsigned

Communication profiles: FF, ISA100

Purpose: Get unsigned value from VARIABLE

The builtin get_unsigned is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

The method associated with pre-edit, post-edit, post-read, or pre-write actions are run when a VARIABLE, which has one of these actions defined as an attribute, is read by an application, a user method, or an edit method.

Lexical structure

```
get_unsigned(value)
```

The lexical elements of the builtin get_unsigned are shown in Table 139.

Table 139 – Builtin get_unsigned

Parameter name	Type	Direction	Usage	Description
value	unsigned long	O	m	The returned value.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long value;
status = get_unsigned(&value);
```

6.4.9 Builtin igetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE

The builtin igetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
igetval(VOID)
```

The lexical element of the builtin igetval is shown in Table 140.

Table 140 – Builtin igetval

Parameter name	Type	Direction	Usage	Description
<return>	long long	O	m	The value of the VARIABLE.

6.4.10 Builtin isetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Set VARIABLE value

The builtin isetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
isetval(value)
```

The lexical elements of the builtin isetval are shown in Table 141.

Table 141 – Builtin isetval

Parameter name	Type	Direction	Usage	Description
value	long long	I	m	The new value of the VARIABLE.
<return>	long long	O	m	The value of the VARIABLE.

6.4.11 Builtin isOffline

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Determine whether the EDD Application can currently communicate with the field device

The builtin isOffline returns a non-zero value when the EDD Application state (within the context of the calling method) is currently "Offline".

Lexical structure

```
isOffline()
```

The lexical element of the builtin isOffline is shown in Table 142.

Table 142 – Builtin isOffline

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	Non-zero if the EDD application is offline and cannot communicate with the device, 0 if the EDD application can communicate with the device.

Example

```
int offline;
offline = isOffline();
```

6.4.12 Builtin lgetval

Communication profiles: CS, GPE, PB, PN

Purpose

The builtin lgetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
lgetval (VOID)
```

The lexical element of the builtin lgetval is shown in Table 143.

Table 143 – Builtin lgetval

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	The value of the VARIABLE.

6.4.13 Builtin lsetval

Communication profiles: CS, GPE, PB, PN

Purpose: Set long value

The builtin lsetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
lsetval (value)
```

The lexical elements of the builtin lsetval are shown in Table 144.

Table 144 – Builtin lsetval

Parameter name	Type	Direction	Usage	Description
value	int		m	The new value of the VARIABLE.
<return>	int	O	m	The value of the VARIABLE.

6.4.14 Builtin put_date

Communication profiles: FF, ISA100

Purpose: Store date value in VARIABLE

The builtin put_date is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
put_date (data, size)
```

The lexical elements of the builtin put_date are shown in Table 145.

Table 145 – Builtin put_date

Parameter name	Type	Direction	Usage	Description
data	char[]	I	m	The new value of the VARIABLE.
size	long	I	m	The length of the new value.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long date_size, status;
char date_value[7];
```

```
/* set DATE_AND_TIME to August 19, 2016 14:30:03 */
```

```
date_size = 7;
date_value[0]=0x0B;      /* 3 seconds = 3000 ms = 0xBB8 */
date_value[1]=0xB8;
date_value[2]= 30;      /* 30 minutes */
date_value[3]= 14;      /* 14 hours */
date_value[4]=0xB3;      /* 0xB3 = (5 (Friday) << 5) + 19 (day of month) */
date_value[5]= 8;      /* August = 8 */
date_value[6]= 116;      /* year of 2016 = 116 + 1900 */
```

```
status = put_date(date_value, date_size);
```

6.4.15 Builtin put_double**Communication profiles: FF, ISA100****Purpose: Store double value to VARIABLE**

The builtin put_double is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
put_double(value)
```

The lexical elements of the builtin put_double are shown in Table 146.

Table 146 – Builtin put_double

Parameter name	Type	Direction	Usage	Description
value	double	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
double value;
value = 3.14159;
status = put_double(value);
```

6.4.16 Builtin put_float

Communication profiles: FF, ISA100

Purpose: Store float value to VARIABLE

The builtin put_float is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

put_float(value)

The lexical elements of the builtin put_float are shown in Table 147.

Table 147 – Builtin put_float

Parameter name	Type	Direction	Usage	Description
value	double	I	m	The new value of the VARIABLE. The value is passed as a double to allow an expression to be passed. Many compilers perform a single to double conversion prior to evaluating expressions.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
float value;
value = 279.55025;
status = put_float((double) value);
```

6.4.17 Builtin put_signed

Communication profiles: FF, ISA100

Purpose: Store signed value to VARIABLE

The builtin put_signed is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type specified in Table 9.

Lexical structure

put_signed(value)

The lexical elements of the builtin put_signed are shown in Table 148.

Table 148 – Builtin put_signed

Parameter name	Type	Direction	Usage	Description
value	long	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
long value;
value = -3592;
status = put_signed(value);

```

6.4.18 Builtin put_string

Communication profiles: FF, ISA100

Purpose: Store string value to VARIABLE

The builtin put_string is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

VARIABLEs with data types of enumerated, bit enumerated, and index shall be handled by the method because they are not supported by the programming language-C.

Lexical structure

```
put_string(string, len)
```

The lexical elements of the builtin put_string are shown in Table 149.

Table 149 – Builtin put_string

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The new value of the VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
len	long	I	m	The length of the new value.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
status = put_string("xyz", 3);

```

6.4.19 Builtin put_unsigned

Communication profiles: FF, ISA100

Purpose: Store unsigned value to VARIABLE

The builtin put_unsigned is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

put_unsigned(value)

The lexical elements of the builtin put_unsigned are shown in Table 150.

Table 150 – Builtin put_unsigned

Parameter name	Type	Direction	Usage	Description
value	unsigned long	I	o	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long value;
value = 123;
status = put_unsigned(value);
```

6.4.20 Builtin sgetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE of a string type

The builtin sgetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin returns the value of a VARIABLE as it was received from the connected device. Custom calculations may then be performed on the VARIABLE value before it is stored in the EDD application. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be retrieved, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

sgetval(value)

The lexical elements of the builtin sgetval are shown in Table 151.

Table 151 – Builtin sgetval

Parameter name	Type	Direction	Usage	Description
value	DD_STRING	O	m	The value of the VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	BI_SUCCESS if the assignment was successful.

Example

```
DD_STRING sval;
sgetval(sval);
```

In the above example (assume that this is in a pre-edit method on the VARIABLE “tag”, an 8 character variable), sval will be set to the current value of the VARIABLE “tag”.

6.4.21 Builtin ssetval

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Set the value of a VARIABLE of a string type

The builtin ssetval is specifically designed to be used in VARIABLE actions. Refer to Table 9 for the list of action types which may call this builtin.

This builtin sets the value of a VARIABLE to the specified value. If the string passed in is longer than the size specified for the VARIABLE, it will be truncated to fit. If the string passed in is shorter than the size specified for the VARIABLE, it will be padded with spaces. The method aborts if it is not an action called from a VARIABLE, if the value of the VARIABLE cannot be set, or if the VARIABLE is not of the appropriate type defined in Table 9.

Lexical structure

```
ssetval(value)
```

The lexical elements of the builtin ssetval are shown in Table 152.

Table 152 – Builtin ssetval

Parameter name	Type	Direction	Usage	Description
value	DD_STRING	I	m	The value of the VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	The value of the VARIABLE.

6.5 Variable access builtins**6.5.1 Builtin assign**

Communication profiles: FF, ISA100

Purpose: Sets the value of a parameter

The builtin assign sets the value of a VARIABLE to another VARIABLE's value. It reads the value of the source VARIABLE (from either the device or the variable table) and sets the value of the destination VARIABLE (to either the device or the variable table) equal to the value of the source VARIABLE.

NOTE The builtin is a shorthand way of doing an `get_<type>_value` followed by a `put_<type>_value` where `<type>` is any one of the following value types: date, float, double, signed, string, unsigned.

The parameters `dst_id`, `src_id`, `dst_member_id`, and `src_member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the reference of the VARIABLE. Both the source VARIABLES and the destination VARIABLE shall have the same type.

Lexical structure

```
assign(dst_id, dst_member_id, src_id, src_member_id)
```

The lexical elements of the builtin `assign` are shown in Table 153.

Table 153 – Builtin assign

Parameter name	Type	Direction	Usage	Description
<code>dst_id</code>	unsigned long	I	m	The item id of the destination VARIABLE.
<code>dst_member_id</code>	unsigned long	I	m	The member id of the destination VARIABLE.
<code>src_id</code>	unsigned long	I	m	The item id of the source VARIABLE.
<code>src_member_id</code>	unsigned long	I	m	The member id of the source VARIABLE.
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = assign(ITEM_ID(PARAM.OUT), MEMBER_ID(VALUE),
```

6.5.2 Builtin assign_double

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type double

The builtin `assign_double` assigns a value to a VARIABLE. The referenced VARIABLE shall be VARIABLE of type DOUBLE.

Lexical structure

```
assign_double(device_var, new_value)
```

The lexical elements of the builtin `assign_double` are shown in Table 154.

Table 154 – Builtin assign_double

Parameter name	Type	Direction	Usage	Description
<code>device_var</code>	reference	I	m	The name of the VARIABLE.
<code>new_value</code>	double	I	m	The new value of the referenced VARIABLE.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. <code>BI_SUCCESS</code> if the assignment was successful, and <code>BI_ERROR</code> if the VARIABLE identifier was invalid or erroneous.

Example

```
assign_double(pressure_upper_range_value, 1.2345678);
```

In this example, the VARIABLE `pressure_upper_range_value` will be set to the float value 1,2345678.

6.5.3 Builtin `assign_float`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type float

The builtin `assign_float` assigns a value to VARIABLE. The referenced VARIABLE shall be a VARIABLE of type FLOAT.

Lexical structure

```
assign_float(device_var, new_value)
```

The lexical elements of the builtin `assign_float` are shown in Table 155.

Table 155 – Builtin `assign_float`

Parameter name	Type	Direction	Usage	Description
<code>device_var</code>	reference	I	m	The name of the VARIABLE.
<code>new_value</code>	double	I	m	The new value of the referenced VARIABLE.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

Example

```
assign_float(pressure_upper_range_value, 1.23);
```

In this example, the VARIABLE `pressure_upper_range_value` will be set to the floating point value 1,23.

6.5.4 Builtin `assign_int`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type integer

The builtin `assign_int` assigns a value to VARIABLE. The referenced VARIABLE shall be a VARIABLE of type INTEGER.

Lexical structure

```
assign_int(device_var, new_value)
```

The lexical elements of the builtin `assign_int` are shown in Table 156.

Table 156 – Builtin `assign_int`

Parameter name	Type	Direction	Usage	Description
<code>device_var</code>	reference	I	m	The name of the VARIABLE.
<code>new_value</code>	int	I	m	The new value of the referenced VARIABLE.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

Example

```
assign_int(pressure_upper_range_value, 1);
```

In this example, the VARIABLE `pressure_upper_range_value` will be set to the integer value 1.

6.5.5 Builtin assign_var

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign value of one VARIABLE to another

The builtin `assign_var` assigns a value of the source VARIABLE to the destination VARIABLE. Both VARIABLES shall have the same type.

Lexical structure

```
assign_var(destination_var, source_var)
```

The lexical elements of the builtin `assign_var` are shown in Table 157.

Table 157 – Builtin assign_var

Parameter name	Type	Direction	Usage	Description
<code>destination_var</code>	reference	I	m	The name of the destination VARIABLE.
<code>source_var</code>	reference	I	m	The name of the source VARIABLE.
<code><return></code>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

Example

```
vassign(pressure_units, range_units);
```

In this example, the current value of `range_units` will be stored in the current value of `pressure_units`.

6.5.6 Builtin assign2

Communication profiles: FF, ISA100

Purpose: Sets the value of a VARIABLE, record member, or array element

The builtin `assign2` reads a value from the source block and writes it to the destination block. In a caching EDD application, the source value is retrieved from the source block's variable table and stored in the destination block's variable table, whereas in a non-caching EDD application the source value is read from the source block in the device and written to the destination block in the device.

The `dst_blk_id`, `dst_blk_num`, `dst_id`, and `dst_member_id` parameters collectively specify the destination VARIABLE, record member, or array element; while the `src_blk_id`, `src_blk_num`, `src_id`, and `src_member_id` parameters collectively specify the source VARIABLE, record member, or array element.

The `dst_blk_id` and `src_blk_id` parameters specify the item id of the blocks to be accessed, while the `dst_blk_num` and `src_blk_num` parameters specify the relative indices of the blocks to be accessed.

If the `dst_id` parameter references a simple VARIABLE, the `dst_member_id` parameter is ignored. If the `dst_id` parameter references a record, the `dst_member_id` parameter specifies a member of the record. If the `dst_id` parameter references an array parameter, the `dst_member_id` parameter specifies an element of the array.

If the `src_id` parameter references a simple VARIABLE, the `src_member_id` parameter is ignored. If the `src_id` parameter references a record, the `src_member_id` parameter specifies a member of the record. If the `src_id` parameter references an array parameter, the `src_member_id` parameter specifies an element of the array.

The `dst_blk_id`, `src_blk_id`, `dst_id`, `src_id`, `dst_member_id`, and `src_member_id` parameters are typically specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `dst_blk_num` and `src_blk_num` parameters are typically specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

NOTE This built-in function provides a shorthand way of performing a `get_<type>_value2()` followed by a `put_<type>_value2()`, where `<typw>` is one of the following: date, float, double, signed, string, unsigned.

Both the source value and the destination value shall have the same type.

Lexical structure

```
assign2(dst_blk_id, dst_blk_num, dst_id, dst_member_id, src_blk_id,
src_blk_num, src_id, src_member_id)
```

The lexical elements of the builtin `assign2` are shown in Table 158.

Table 158 – Builtin assign2

Parameter name	Type	Direction	Usage	Description
<code>dst_blk_id</code>	unsigned long	I	m	The item id of the destination block.
<code>dst_blk_num</code>	unsigned long	I	m	The occurrence number of the destination block.
<code>dst_id</code>	unsigned long	I	m	The item id of the destination VARIABLE, record, or array.
<code>dst_member_id</code>	unsigned long	I	m	The member id of the destination record member or array element.
<code>src_blk_id</code>	unsigned long	I	m	The item id of the source block.
<code>src_blk_num</code>	unsigned long	I	m	The occurrence number of the source block.
<code>src_id</code>	unsigned long	I	m	The item id of the source VARIABLE, record, or array.
<code>src_member_id</code>	unsigned long	I	m	The member id of the source record member or array element.
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long aiBlock;
aiBlock = ITEM_ID(__analog_input_block);
status = assign2(aiBlock, 0, MEMBER_ID(OUT), MEMBER_ID(VALUE),
aiBlock, 0, MEMBER_ID(PV), MEMBER_ID(VALUE));
```

6.5.7 Builtin dassign

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type double

The builtin dassign assigns a value to a VARIABLE. The referenced VARIABLE shall be a VARIABLE of type DOUBLE.

Lexical structure

```
dassign(device_var, new_value)
```

The lexical elements of the builtin dassign are shown in Table 159.

Table 159 – Builtin dassign

Parameter name	Type	Direction	Usage	Description
device_var	reference	I	m	The name of the VARIABLE.
new_value	double	I	m	The new value of the referenced VARIABLE.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if the VARIABLE identifier was invalid or erroneous.

6.5.8 Builtin fassign

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type float

The builtin fassign assigns a value to a VARIABLE. The referenced VARIABLE shall be a VARIABLE of type FLOAT.

Lexical structure

```
fassign(device_var, new_value)
```

The lexical elements of the builtin fassign are shown in Table 160.

Table 160 – Builtin fassign

Parameter name	Type	Direction	Usage	Description
device_var	reference	I	m	The name of the VARIABLE.
new_value	double	I	m	The new value of the referenced VARIABLE.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

6.5.9 Builtin float_value

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE of type float

The builtin float_value returns the value of the specified VARIABLE. The referenced VARIABLE shall be of type FLOAT.

Lexical structure

```
float_value(source_var_name)
```

The lexical elements of the builtin float_value are shown in Table 161.

Table 161 – Builtin float_value

Parameter name	Type	Direction	Usage	Description
source_var_name	reference	I	m	The name of a VARIABLE reference.
<return>	double	O	m	The value of the VARIABLE.

Example

```
float fval;
fval = float_value(pressure_upper_range_value);
```

In this example, the value of pressure_upper_range_value is returned and stored in the method-variable fval.

6.5.10 Builtin fvar_value

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE of type float

The builtin fvar_value returns the value of the specified VARIABLE. The referenced VARIABLE shall be of type FLOAT.

Lexical structure

```
fvar_value(source_var)
```

The lexical elements of the builtin fvar_value are shown in Table 162.

Table 162 – Builtin fvar_value

Parameter name	Type	Direction	Usage	Description
source_var	reference	I	m	The name of the VARIABLE reference.
<return>	double	O	m	The value of the VARIABLE.

6.5.11 Builtin get_date_value

Communication profiles: FF, ISA100

Purpose: Gets the current value of a date VARIABLE

The builtin get_date_value retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The value is an octet buffer that may contain up to eight characters and a VARIABLE of device type date-and-time, time, or duration.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE in question.

Lexical structure

```
get_date_value(id, member_id, data, size)
```

The lexical elements of the builtin get_date_value are shown in Table 163.

Table 163 – Builtin get_date_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	An id of a VARIABLE reference.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
data	char[]	O	m	A buffer of max. 8 octets for the result of the builtin.
size	long	O	m	The size of the used buffer.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status, date_size;
char date_buf[8];
```

```
status = get_date_value(ITEM_ID(PARAM.CALIBRATION), MEMBER_ID(DATE),
date_buf, &date_size);
```

6.5.12 Builtin get_date_value2**Communication profiles: FF, ISA100****Purpose: Retrieves the value of a VARIABLE, record member or array element**

The builtin get_date_value2 retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The value is an octet buffer that may contain up to eight characters and contain a VARIABLE of device type date-and-time, time, or duration.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The blk_id, id, and member_id parameters are typically specified using the builtins ITEM_ID and MEMBER_ID.

The blk_num parameter is typically specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The value and length parameters specify a buffer that, upon return, contains up to eight characters with a data type of Date, Time of Day, or Time Difference.

The type of the specified VARIABLE, record member or array element shall be one of the following:

- DATE_AND_TIME
- TIME
- DURATION

Lexical structure

```
get_date_value2(blk_id, blk_num, id, member_id, data, size)
```

The lexical elements of the builtin `get_date_value2` are shown in Table 164.

Table 164 – Builtin `get_date_value2`

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	An id of a VARIABLE, RECORD, or ARRAY reference.
member_id	unsigned long	I	m	The record member or array element to be accessed.
data	char[]	O	m	A buffer of max. 8 octets for the result of the builtin.
size	long	O	m	The size of the used buffer.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status, date_size;
char date_buf[8];
```

```
status = get_date_value2(ITEM_ID(transducer_block), 0,
MEMBER_ID(CALIBRATION), MEMBER_ID(TEMP), date_buf, &date_size);
```

6.5.13 Builtin `get_double_value`

Communication profiles: FF, ISA100

Purpose: Gets the current value of a double float VARIABLE

The builtin `get_double_value` retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the VARIABLE.

The VARIABLE shall have a data type of DOUBLE.

Lexical structure

```
get_double_value(id, member_id, value)
```

The lexical elements of the builtin `get_double_value` are shown in Table 165.

Table 165 – Builtin get_double_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	double	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
double value;
status = get_double_value(ITEM_ID(PARAM.SAMPLE), 0, &value);
```

6.5.14 Builtin get_double_value2**Communication profiles: FF, ISA100****Purpose: Gets the current value of a VARIABLE, record member or array element**

The builtin `get_double_value2` retrieves the current scaled value of the specified VARIABLE, record member or array element from the cache or device and returns it.

The `blk_id` parameter specifies the item id of the block to be accessed, while the `blk_num` parameter specifies the occurrence number of the block to be accessed.

If the `id` parameter references a simple VARIABLE, the `member_id` parameter is ignored. If the `id` parameter references a record, the `member_id` parameter specifies a member of the record. If the `id` parameter references an array parameter, the `member_id` parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the VARIABLE. The `blk_num` parameter is typically specified using the Builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The VARIABLE, record member or array shall have a data type of DOUBLE.

Lexical structure

```
get_double_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin `get_double_value2` are shown in Table 166.

Table 166 – Builtin get_double_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The Item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	double	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
double value;
status = get_double_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, &value);
```

6.5.15 Builtin get_float_value**Communication profiles: FF, ISA100****Purpose: Gets the current scaled value of a floating-point VARIABLE**

The builtin get_float_value retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The parameters id and member id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE.

The VARIABLE shall have a data type of FLOAT.

Lexical structure

```
get_float_value(id, member_id, value)
```

The lexical elements of the builtin get_float_value are shown in Table 167.

Table 167 – Builtin get_float_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	float	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
float value;
status = get_float_value(ITEM_ID(PARAM.SAMPLE), 0, &value);
```

6.5.16 Builtin get_float_value2

Communication profiles: FF, ISA100

Purpose: Retrieves the value of a VARIABLE, record member or array element

The builtin get_float_value2 retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE. The blk_num parameter is typically specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The type of the specified VARIABLE, record member or array element shall be FLOAT.

Lexical structure

```
get_float_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin get_float_value2 are shown in Table 168.

Table 168 – Builtin get_float_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	float	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
float value;
status = get_float_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, &value);
```

6.5.17 Builtin get_signed_value

Communication profiles: FF, ISA100

Purpose: Gets the current scaled value of a signed integer VARIABLE

The builtin get_signed_value retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE. The VARIABLE shall have a data type of signed long.

Lexical structure

get_signed_value(id, member_id, value)

The lexical elements of the builtin get_signed_value are shown in Table 169.

Table 169 – builtin get_signed_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	signed long	O	m	The returned value of the signed VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
long value;
status = get_signed_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(DECIMAL_PT),
&value);
```

6.5.18 Builtin get_signed_value2

Communication profiles: FF, ISA100

Purpose: Retrieves the value of a VARIABLE, record member or array element

The builtin get_signed_value2 retrieves the current scaled value of the specified VARIABLE, record member or array element from the cache or device and returns it.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID. The Blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The VARIABLE, record member or array element shall have a data type of signed long.

Lexical structure

```
get_signed_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin `get_signed_value2` are shown in Table 170.

Table 170 – Builtin `get_signed_value2`

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	signed long	O	m	The returned value of the signed VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
long value;
status = get_signed_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_SCALE), MEMBER_ID(DECIMAL_PT), &value);
```

6.5.19 Builtin `get_string_value`

Communication profiles: FF, ISA100

Purpose: Gets the current value of a string VARIABLE

The builtin `get_string_value` retrieves the current string value of the specified VARIABLE from the cache or device and returns it.

The id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the variable in question.

The VARIABLE shall have a data type of char and be a byte buffer that can contain various non-ANSI data types such as:

- VISIBLE
- OCTET
- PASSWORD
- BITSTRING
- EUC
- ASCII

Lexical structure

```
get_string_value(id, member_id, string, len)
```

The lexical elements of the builtin `get_string_value` are shown in Table 171.

Table 171 – Builtin get_string_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The id of a VARIABLE reference.
member_id	unsigned long	I	m	The member id of a BLOCK_A, PARAMETER_LIST, RECORD, VALUE_ARRAY, COLLECTION or REFERENCE_ARRAY.
string	char*	O	m	The buffer of result string.
len	long	O	m	The size of the result string.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
long len;
char buf[256];
len = 256;
status = get_string_value(ITEM_ID(PARAM.TAG_DESC), 0, buf, &len);

```

6.5.20 Builtin get_string_value2**Communication profiles: FF, ISA100****Purpose: Retrieves the value of a VARIABLE, record member or array element**

The builtin get_string_value2 retrieves the current string value of the specified VARIABLE, record member or array element from the cache or device and returns it.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID. The blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The type of the specified VARIABLE, record member or array element shall be one of the following:

- EUC
- ASCII
- PASSWORD
- BITSTRING
- VISIBLE
- OCTET

Lexical structure

```
get_string_value2(blk_id, blk_num, id, member_id, string, len)
```

The lexical elements of the builtin get_string_value2 are shown in Table 172.

Table 172 – Builtin get_string_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The id of a VARIABLE reference.
member_id	unsigned long	I	m	The member id of a BLOCK_A, PARAMETER_LIST, RECORD, VALUE_ARRAY, COLLECTION or REFERENCE_ARRAY.
string	char*	O	m	Specifies the buffer of result string.
len	long	O	m	The size of the result string.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
long len;
char buf[256];
len = 256;
status = get_string_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(TAG_DESC), 0, buf, &len);

```

6.5.21 Builtin get_unsigned_value**Communication profiles: FF, ISA100****Purpose: Gets the current value of an unsigned integer VARIABLE**

The builtin get_unsigned_value retrieves the current scaled value of the specified VARIABLE from the cache or device and returns it.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE in question.

The VARIABLE shall have a data type of unsigned. An unsigned value may contain VARIABLES of the following data types:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

VARIABLEs with data types of enumerated, bit_enumerated, and index shall be handled by the method because they are not supported by the programming language-C.

Lexical structure

```
get_unsigned_value(id, member_id, value)
```

The lexical elements of the builtin get_unsigned_value are shown in Table 173.

Table 173 – Builtin get_unsigned_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	unsigned long	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long value;
status = get_unsigned_value (ITEM_ID (PARAM.PV_STATE), 0, &value);

```

6.5.22 Builtin get_unsigned_value2**Communication profiles: FF, ISA100****Purpose: Retrieves the value of a VARIABLE, record member or array element**

The builtin get_unsigned_value2 retrieves the current scaled value of the specified VARIABLE, record member or array element from the cache or device and returns it.

The blk_id parameter specifies the item id of the block to be accessed, while the blockInstance parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID. The blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The VARIABLE shall have a data type of unsigned. An unsigned value may contain VARIABLES of the following data types:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

VARIABLEs with data types of enumerated, bit_enumerated, and index shall be handled by the method because they are not supported by the programming language-C.

Lexical structure

```
get_unsigned_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin get_unsigned_value2 are shown in Table 174.

Table 174 – Builtin get_unsigned_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
value	unsigned long	O	m	The returned value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long value;
status = get_unsigned_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_STATE), 0, &value);
```

6.5.23 Builtin iassign

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign a value to a VARIABLE of type integer

The builtin iassign will assign the specified value to the VARIABLE. The VARIABLE shall reference a VARIABLE of type INTEGER.

Lexical structure

```
iassign(device_var, new_value)
```

The lexical elements of the builtin iassign are shown in Table 175.

Table 175 – Builtin iassign

Parameter name	Type	Direction	Usage	Description
device_var	reference	I	m	The name of the VARIABLE to access.
new_value	int	I	m	The new value of the VARIABLE.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

6.5.24 Builtin int_value

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE of type integer

The builtin int_value will return the value of the specified VARIABLE. The VARIABLE shall be of type INTEGER.

Lexical structure

```
int_value(variable)
```

The lexical elements of the builtin `int_value` are shown in Table 176.

Table 176 – Builtin `int_value`

Parameter name	Type	Direction	Usage	Description
variable	reference	I	m	The VARIABLE whose value shall be retrieved.
<return>	int	O	m	The value of the VARIABLE.

6.5.25 Builtin `ivar_value`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get the value of a VARIABLE of type integer

The builtin `ivar_value` will return the value of the specified VARIABLE. The VARIABLE `source_var_name` shall be of type INTEGER.

Lexical structure

```
ivar_value(source_var_name)
```

The lexical elements of the builtin `ivar_value` are shown in Table 177.

Table 177 – Builtin `ivar_value`

Parameter name	Type	Direction	Usage	Description
source_var_name	reference	I	m	The name of the VARIABLE.
<return>	int	O	m	The value of the VARIABLE.

6.5.26 Builtin `lassign`

Communication profiles: CS, GPE, PB, PN

Purpose: Assign long value to VARIABLE

The builtin `lassign` will assign the specified value to the VARIABLE. The VARIABLE shall be reference a VARIABLE of type long.

Lexical structure

```
lassign(var_name, new_value)
```

The lexical elements of the builtin `lassign` are shown in Table 178.

Table 178 – Builtin `lassign`

Parameter name	Type	Direction	Usage	Description
var_name	reference	I	m	The name of the VARIABLE.
new_value	long	I	m	The new value of the VARIABLE.
<return>	int	O	m	One of the return codes specified in Table 377.

6.5.27 Builtin long_value

Communication profiles: CS, GPE, PB, PN

Purpose: Get long value of VARIABLE

The builtin long_value will return the value of the specified VARIABLE. The VARIABLE source_var_name shall reference a VARIABLE of type long.

Lexical structure

```
long_value(source_var_name)
```

The lexical elements of the builtin long_value are shown in Table 179.

Table 179 – Builtin long_value

Parameter name	Type	Direction	Usage	Description
source_var_name	reference	I	m	The name of the VARIABLE.
<return>	int	O	m	The value of the VARIABLE.

6.5.28 Builtin lvar_value

Communication profiles: CS, GPE, PB, PN

Purpose: Get long value

The builtin lvar_value will return the value of the specified VARIABLE.

Lexical structure

```
lvar_value(source_var_name)
```

The lexical elements of the builtin lvar_value are shown in Table 180.

Table 180 – Builtin lvar_value

Parameter name	Type	Direction	Usage	Description
source_var_name	reference	I	m	The name of the VARIABLE.
<return>	long	O	m	The value of the VARIABLE.

6.5.29 Builtin put_date_value

Communication profiles: FF, ISA100

Purpose: Puts the specified value of a date_and_time, time, or duration

The builtin put_date_value stores the provided value of the specified scaled VARIABLE into the cache or device.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE in question.

It puts the specified value of the date_and_time, time, or duration VARIABLE into the cache or device.

The VARIABLE shall have a data type of char. The char string may contain up to eight chars and contain a VARIABLE of the following data types:

- DATE_AND_TIME
- TIME
- DURATION

Lexical structure

put_date_value(id, member_id, data, size)

The lexical elements of the builtin put_date_value are shown in Table 181.

Table 181 – Builtin put_date_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
data	char[]	I	m	The new value of the VARIABLE.
size	long	I	m	The length of the new value.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long duration_size, status;
char duration_value[6];

/* set DURATION to 25 (days): 10 (hours): 20: 05.218 */

duration_size = 6;
duration_value[0] = 0x02; /*0x0237B4E2 = 37205218 ms since midnight */
duration_value[1] = 0x37;
duration_value[2] = 0xB4;
duration_value[3] = 0xE2;
duration_value[4] = 0x00; /*0x0019 = 25 days */
duration_value[5] = 0x19;

status = put_date_value(ITEM_ID(PARAM.CALIBRATION), MEMBER_ID(DURATION),
duration_value, duration_size);
```

6.5.30 Builtin put_date_value2

Communication profiles: FF, ISA100

Purpose: Sets the value of a VARIABLE, record member or array element

The builtin put_date_value2 stores the provided value of the specified scaled VARIABLE into the cache or device.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `blk_num` parameter is specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag` built-in functions.

The `VARIABLE`, record member or array element shall have a data type of `char`. The `char` string may contain up to eight chars and contain a `VARIABLE` of the following data types:

- `DATA_AND_TIME`
- `TIME`
- `DURATION`

Lexical structure

```
put_date_value2(blk_id, blk_num, id, member_id, data, size)
```

The lexical elements of the builtin `put_date_value2` are shown in Table 182.

Table 182 – Builtin `put_date_value2`

Parameter name	Type	Direction	Usage	Description
<code>blk_id</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> instance to access.
<code>blk_num</code>	unsigned long	I	m	The occurrence number of the <code>BLOCK_A</code> instance to access.
<code>id</code>	unsigned long	I	o	The item id of the <code>VARIABLE</code> to access.
<code>member_id</code>	unsigned long	I	o	The member id of the <code>VARIABLE</code> to access.
<code>data</code>	<code>char[]</code>	I	m	The new value of the <code>VARIABLE</code> .
<code>size</code>	long	I	m	The length of the new value.
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status, time_size;
char time[6];

/* set TIME to Oct 14, 2010 03:03:03 */

time_size = 6;
time[0]=0; /* 0xA79658 = 10983000 ms since midnight */
time[1]=0xA7;
time[2]=0x96;
time[3]=0x58;
time[4]=0x26; /* 0x2637 = 9783 days since January 1, 1984 */
time[5]=0x37;

status = put_date_value2(ITEM_ID(transducer_block), 0,
MEMBER_ID(CALIBRATION), MEMBER_ID(TIME), time, time_size);
```

6.5.31 Builtin `put_double_value`

Communication profiles: FF, ISA100

Purpose: Puts the specified scaled value of a double `VARIABLE`

The builtin `put_double_value` stores the provided value of the specified scaled `VARIABLE` into the cache or device.

The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the `VARIABLE` in question.

The `VARIABLE` shall have a data type of `DOUBLE`.

Lexical structure

```
put_double_value(id, member_id, value)
```

The lexical elements of the builtin `put_double_value` are shown in Table 183.

Table 183 – Builtin `put_double_value`

Parameter name	Type	Direction	Usage	Description
<code>id</code>	unsigned long	I	o	The item id of the <code>VARIABLE</code> to access.
<code>member_id</code>	unsigned long	I	o	The member id of the <code>VARIABLE</code> to access.
<code>value</code>	double	I	m	The new value of the <code>VARIABLE</code> .
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
double value;
value = 100090.738;
status = put_double_value(ITEM_ID(PARAM_SAMPLE), 0, value);
```

6.5.32 Builtin `put_double_value2`

Communication profiles: FF, ISA100

Purpose: Sets the value of a `VARIABLE`, record member or array element

The builtin `put_double_value2` stores the provided value of the specified scaled `VARIABLE`, record member or array element into the cache or device.

The `blk_id` parameter specifies the item id of the block to be accessed, while the `blk_num` parameter specifies the occurrence number of the block to be accessed.

If the `id` parameter references a simple `VARIABLE`, the `member_id` parameter is ignored. If the `id` parameter references a record, the `member_id` parameter specifies a member of the record. If the `id` parameter references an array parameter, the `member_id` parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `blk_num` parameter is specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The `VARIABLE`, record member or array element shall have a data type of `DOUBLE`.

Lexical structure

```
put_double_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin `put_double_value2` are shown in Table 184.

Table 184 – Builtin put_double_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	double	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
double value;
value = 1000090.738;
status = put_double_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, value);

```

6.5.33 Builtin put_float_value**Communication profiles: FF, ISA100****Purpose: Puts the specified value of a floating-point VARIABLE**

The builtin put_float_value stores the provided value of the specified scaled VARIABLE in the cache or the device.

The type of value is double to accommodate automatic promotion, which could occur whenever two floating-point values are multiplied.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID builtins and the name or reference of the VARIABLE in question.

The VARIABLE shall have a data type of FLOAT.

Lexical structure

```
put_float_value(id, member_id, value)
```

The lexical elements of the builtin put_float_value are shown in Table 185.

Table 185 – Builtin put_float_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	double	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
float value;
value = 57.2958;
status = put_float_value2(ITEM_ID(PARAM.SAMPLE), 0, (double) value);

```

6.5.34 Builtin put_float_value2**Communication profiles: FF, ISA100****Purpose: Sets the value of a VARIABLE, record member or array element**

The builtin put_float_value2 stores the provided value of the specified scaled VARIABLE, record member or array element in the cache or the device.

The type of value is double to accommodate automatic promotion, which could occur whenever two floating-point values are multiplied.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID builtins. The blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The VARIABLE shall have a data type of FLOAT.

Lexical structure

```
put_float_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin put_float_value2 are shown in Table 186.

Table 186 – Builtin put_float_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	double	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
float value;
value = 57.2958;
status = put_float_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, (double) value);

```

6.5.35 Builtin put_signed_value**Communication profiles: FF, ISA100****Purpose: Puts the specified value of a signed integer VARIABLE**

The builtin put_signed_value stores the new value of the specified scaled VARIABLE into the cache or device.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE in question.

The VARIABLE to be written shall have a long data type.

Lexical structure

```
put_signed_value(id, member_id, value)
```

The lexical elements of the builtin put_signed_value are shown in Table 187.

Table 187 – Builtin put_signed_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	long	I	m	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
status = put_signed_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(DECIMAL_PT),
3);

```

6.5.36 Builtin put_signed_value2**Communication profiles: FF, ISA100****Purpose: Sets the value of a VARIABLE, record member or array element**

The builtin put_signed_value2 stores the new value of the specified scaled VARIABLE, record member or array element into the cache or device.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the `id` parameter references a simple `VARIABLE`, the `member_id` parameter is ignored. If the `id` parameter references a record, the `member_id` parameter specifies a member of the record. If the `id` parameter references an array parameter, the `member_id` parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `blk_num` parameter is specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The `VARIABLE`, record member or array element to be written shall have a long data type.

Lexical structure

```
put_signed_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin `put_signed_value2` are shown in Table 188.

Table 188 – Builtin `put_signed_value2`

Parameter name	Type	Direction	Usage	Description
<code>blk_id</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> instance to access.
<code>blk_num</code>	unsigned long	I	m	The occurrence number of the <code>BLOCK_A</code> instance to access.
<code>id</code>	unsigned long	I	o	The item id of the <code>VARIABLE</code> to access.
<code>member_id</code>	unsigned long	I	o	The member id of the <code>VARIABLE</code> to access.
<code>value</code>	long	I	m	The new value of the <code>VARIABLE</code> .
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = put_signed_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_SCALE), MEMBER_ID(DECIMAL_PT), 3);
```

6.5.37 Builtin `put_string_value`

Communication profiles: FF, ISA100

Purpose: Sets the value of a string `VARIABLE`

The builtin `put_string_value` stores the provided value of the specified `VARIABLE` into the cache or device.

The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the `VARIABLE` in question.

The `VARIABLE` shall have a data type of `char`; a `char` string may contain `VARIABLES` of the following data types:

- ASCII
- PASSWORD
- EUC (extended unicode)
- BITSTRING

- VISIBLE
- OCTET

VARIABLEs with data types of enumerated, bit_enumerated, and index shall be handled by the method because they are not supported by the programming language-C.

Lexical structure

```
put_string_value(id, member_id, string, len)
```

The lexical elements of the builtin put_string_value are shown in Table 189.

Table 189 – Builtin put_string_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	l	o	The item id of the VARIABLE to access.
member_id	unsigned long	l	o	The member id of the VARIABLE to access.
string	DD_STRING	l	m	The new value of the VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
len	long	l	m	The length of the new value.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = put_string_value(ITEM_ID(PARAM.TAG_DESC), 0, "|en|PID-101", 11);
```

6.5.38 Builtin put_string_value2

Communication profiles: FF, ISA100

Purpose: Sets the value of a VARIABLE, record member, or array element

The builtin put_string_value2 stores the provided value of the specified VARIABLE, record member or array element into the cache or device.

The blk_id parameter specifies the item id of the block to be accessed, while the blk_num parameter specifies the occurrence number of the block to be accessed.

If the id parameter references a simple VARIABLE, the member_id parameter is ignored. If the id parameter references a record, the member_id parameter specifies a member of the record. If the id parameter references an array parameter, the member_id parameter specifies an element of the array.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID. The blk_num parameter is specified using the builtins get_block_instance_by_object_index and get_block_instance_by_block_tag.

The VARIABLE, record member or array element shall have a data type of char; a char string may contain VARIABLEs of the following data types:

- ASCII
- PASSWORD

- EUC (extended unicode)
- BITSTRING
- VISIBLE
- OCTET

VARIABLEs, record members or array elements with data types of enumerated, bit_enumerated, and index shall be handled by the method because they are not supported by programming language-C.

Lexical structure

```
put_string_value2(blk_id, blk_num, id, member_id, string, len)
```

The lexical elements of the builtin put_string_value2 are shown in Table 190.

Table 190 – Builtin put_string_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	l	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	l	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	l	o	The item id of the VARIABLE, record, or array to access.
member_id	unsigned long	l	o	The member id of the VARIABLE, record member, or array element to access.
string	DD_STRING	l	m	The new value of the VARIABLE, record member, or array element. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
len	long	l	m	The length of the new value.
<return>	long	o	m	One of the return codes specified in Table 376.

Example

```
long status;
status = put_string_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(TAG_DESC), 0, "|en|PID-101", 11);
```

6.5.39 Builtin put_unsigned_value

Communication profiles: FF, ISA100

Purpose: Sets the value of an unsigned VARIABLE

The builtin put_unsigned_value stores the provided value of the specified scaled VARIABLE into the cache or device.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the VARIABLE in question.

The VARIABLE shall have a data type of unsigned. An unsigned value may contain VARIABLEs of the following data types:

- UNSIGNED
- ENUMERATED

- BIT_ENUMERATED
- INDEX
- BOOLEAN

VARIABLEs of these data types shall be handled by the method because they are not supported by the programming language-C. The variable shall be valid in the cache or device.

Lexical structure

```
put_unsigned_value(id, member_id, value)
```

The lexical elements of the builtin `put_unsigned_value` are shown in Table 191.

Table 191 – Builtin `put_unsigned_value`

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	unsigned long	I	o	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long value;
value = 0xFEEE;
status = put_unsigned_value(ITEM_ID(PARAM.PV_STATE), 0, value);
```

6.5.40 Builtin `put_unsigned_value2`

Communication profiles: FF, ISA100

Purpose: Sets the value of a VARIABLE, record member or array element

The builtin `put_unsigned_value2` stores the provided value of the specified scaled VARIABLE into the cache or device.

The `blk_id` parameter specifies the item id of the block to be accessed, while the `blk_num` parameter specifies the occurrence number of the block to be accessed.

If the `id` parameter references a simple VARIABLE, the `member_id` parameter is ignored. If the `id` parameter references a record, the `member_id` parameter specifies a member of the record. If the `id` parameter references an array parameter, the `member_id` parameter specifies an element of the array.

The parameters `blk_id`, `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`. The `blk_num` parameter is specified using the builtins `get_block_instance_by_object_index` and `get_block_instance_by_block_tag`.

The VARIABLE, record member or array element shall have a data type of unsigned. An unsigned value may contain VARIABLEs of the following data types:

- UNSIGNED
- ENUMERATED

- BIT_ENUMERATED
- INDEX
- BOOLEAN

VARIABLEs, of these data types shall be handled by the method because they are not supported by the programming language-C.

Lexical structure

```
put_unsigned_value2(blk_id, blk_num, id, member_id, value)
```

The lexical elements of the builtin `put_unsigned_value2` are shown in Table 192.

Table 192 – Builtin `put_unsigned_value2`

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A instance to access.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A instance to access.
id	unsigned long	I	o	The item id of the VARIABLE to access.
member_id	unsigned long	I	o	The member id of the VARIABLE to access.
value	unsigned long	I	o	The new value of the VARIABLE.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long value;
value = 0xFEEE;
status = put_unsigned_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_STATE), 0, value);
```

6.5.41 Builtin `ret_double_value`

Communication profiles: FF, ISA100

Purpose: Gets the current value of a double VARIABLE

The builtin `ret_double_value` calls the builtin `get_double_value` and returns the value.

Lexical structure

```
ret_double_value(id, member_id)
```

The lexical elements of the builtin `ret_double_value` are shown in Table 193.

Table 193 – Builtin ret_double_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE.
member_id	unsigned long	I	m	The member id of the VARIABLE.
<return>	double	O	m	The specified value.

Example

```
double value;
value = ret_double_value (ITEM_ID (PARAM.SAMPLE), 0);
```

6.5.42 Builtin ret_double_value2**Communication profiles: FF, ISA100****Purpose: Retrieves the value of a VARIABLE, record member or array element**

The builtin ret_double_value2 calls the builtin get_double_value2 and returns the value.

Lexical structure

```
ret_double_value2 (blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin ret_double_value2 are shown in Table 194.

Table 194 – Builtin ret_double_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the block to be accessed.
blk_num	unsigned long	I	m	The occurrence number of the block to be accessed.
id	unsigned long	I	m	The item id of the VARIABLE, record, or array to be accessed.
member_id	unsigned long	I	m	The member id of the record member or array element to be accessed.
<return>	double	O	m	The specified value.

Example

```
double value;
value = ret_double_value2 (ITEM_ID (__analog_input_block), 0,
MEMBER_ID (SAMPLE), 0);
```

6.5.43 Builtin ret_float_value**Communication profiles: FF, ISA100****Purpose: Gets the current value of a float VARIABLE**

The builtin ret_float_value calls the builtin get_float_value and returns the value.

Lexical structure

```
ret_float_value (id, member_id)
```

The lexical elements of the builtin `ret_float_value` are shown in Table 195.

Table 195 – Builtin `ret_float_value`

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE.
member_id	unsigned long	I	m	The member id of the VARIABLE.
<return>	float	O	m	The specified value.

Example

```
float value;
value = ret_float_value(ITEM_ID(PARAM.SAMPLE), 0);
```

6.5.44 Builtin `ret_float_value2`

Communication profiles: FF, ISA100

Purpose: Retrieves the value of a VARIABLE, record member or array element

The builtin `ret_float_value2` calls the builtin `get_float_value2` and returns the value.

Lexical structure

```
ret_float_value2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin `ret_float_value2` are shown in Table 196.

Table 196 – Builtin `ret_float_value2`

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the block to be accessed.
blk_num	unsigned long	I	m	The occurrence number of the block to be accessed.
id	unsigned long	I	m	The item id of the VARIABLE, record, or array to be accessed.
member_id	unsigned long	I	m	The member id of the record member or array element to be accessed.
<return>	float	O	m	The specified value.

Example

```
float value;
value = ret_float_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0);
```

6.5.45 Builtin `ret_signed_value`

Communication profiles: FF, ISA100

Purpose: Gets the current value of a signed integer VARIABLE

The builtin `ret_signed_value` calls the builtin `get_signed_value` and returns the value.

Lexical structure

```
ret_signed_value(id, member_id)
```

The lexical elements of the builtin `ret_signed_value` are shown in Table 197.

Table 197 – Builtin `ret_signed_value`

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE.
member_id	unsigned long	I	m	The member id of the VARIABLE.
<return>	long	O	m	The specified value.

Example

```
long value;
value = ret_signed_value(ITEM_ID(PARAM.SAMPLE), 0);
```

6.5.46 Builtin `ret_signed_value2`

Communication profiles: FF, ISA100

Purpose: Retrieves the value of a VARIABLE, record member or array element

The builtin `ret_signed_value2` calls the builtin `get_signed_value2` and returns the value.

Lexical structure

```
ret_signed_value2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin `ret_signed_value2` are shown in Table 198.

Table 198 – Builtin `ret_signed_value2`

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the block to be accessed.
blk_num	unsigned long	I	m	The occurrence number of the block to be accessed.
id	unsigned long	I	m	The item id of the VARIABLE, record, or array to be accessed.
member_id	unsigned long	I	m	The member id of the record member or array element to be accessed.
<return>	long	O	m	The specified value.

Example

```
long value;
value = ret_signed_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0);
```

6.5.47 Builtin ret_unsigned_value

Communication profiles: FF, ISA100

Purpose: Gets the current value of a unsigned integer VARIABLE

The builtin ret_unsigned_value calls the builtin get_unsigned_value and returns the value.

Lexical structure

```
ret_unsigned_value(id, member_id)
```

The lexical elements of the builtin ret_unsigned_value are shown in Table 199.

Table 199 – Builtin ret_unsigned_value

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE.
member_id	unsigned long	I	m	The member id of the VARIABLE.
<return>	unsigned long	O	m	The specified value.

Example

```
unsigned long value;
value = ret_unsigned_value(ITEM_ID(PARAM.SAMPLE), 0);
```

6.5.48 Builtin ret_unsigned_value2

Communication profiles: FF, ISA100

Purpose: Retrieves the value of a VARIABLE, record member or array element

The builtin ret_unsigned_value2 calls the builtin get_unsigned_value2 and returns the value.

Lexical structure

```
ret_unsigned_value2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin ret_unsigned_value2 are shown in Table 200.

Table 200 – Builtin ret_unsigned_value2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the block to be accessed.
blk_num	unsigned long	I	m	The occurrence number of the block to be accessed.
id	unsigned long	I	m	The item id of the VARIABLE, record, or array to be accessed.
member_id	unsigned long	I	m	The member id of the record member or array element to be accessed.
<return>	unsigned long	O	m	The specified value.

Example

```
unsigned long value;
value = ret_unsigned_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0);
```

6.5.49 Builtin vassign

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Assign value of one VARIABLE to another

The builtin vassign will assign the value of the source VARIABLE to the destination VARIABLE. Both VARIABLES shall have the same type.

Lexical structure

```
vassign(dest_var, source_var)
```

The lexical elements of the builtin vassign are shown in Table 201.

Table 201 – Builtin vassign

Parameter name	Type	Direction	Usage	Description
dest_var	reference	I	m	The identifier of the destination VARIABLE.
source_var	reference	I	m	The identifier of the source VARIABLE.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the assignment was successful, and BI_ERROR if either VARIABLE identifier was invalid or erroneous.

6.6 Normal termination builtins**6.6.1 Builtin discard_on_exit**

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Discard any VARIABLE changes made in a method when the method exits

The builtin discard_on_exit sets the termination action to discard any changes made by the method to VARIABLE values stored in a cache, which have not already been sent to the device. It causes the method to discard any changes to VARIABLE values in the cache when the method ends, and overrides any calls to save_values().

When a method modifies a VARIABLE value in a caching EDD application, it is changing a resident copy of that value. This change is effective only for the life of the method. To change the value(s) in the device, the method shall send the value(s) to the device or call send_on_exit. To save the values only in the cache (rather than send them to the device), call the builtin save_on_exit.

After the method termination, the changed values in the cache return to the device values that existed before they were changed by the method.

All values that have been changed but not sent to the device are handled at the termination of the method by either discarding them, sending them to the device, or saving the changes in a way that survives the end of the method (but does not affect the values in the device).

This builtin takes no action in a non-caching EDD application because all parameter values are written directly to the device.

Communication profiles: FF, HART, ISA100

If a method does not call builtin `discard_on_exit`, `save_on_exit`, or `send_on_exit` prior to exiting properly, any changes made by the method are discarded as if `discard_on_exit` was called.

Communication profiles: CS, GPE, PB, PN

If a method does not call builtin `discard_on_exit`, `save_on_exit`, or `send_on_exit` prior to exiting properly, any changes made by the method are saved as if `save_on_exit` was called.

Lexical structure

`discard_on_exit(VOID)`

The lexical element of the builtin `discard_on_exit` is shown in Table 202.

Table 202 – Builtin `discard_on_exit`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
discard_on_exit();
```

6.6.2 Builtin `save_on_exit`

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Save modified values

The builtin `save_on_exit` sets the termination action to save any changes made by the method to variables in the variable table not already sent to the device. The values in the device are not affected. As long as the application has not changed the values, the values are saved as if written by an application.

When a method modifies a variable's value in the variable table, the value is not written to the device until the method terminates. The changes to the variable table are effective only for the life of the method. To make the values permanent in the variable table, the method shall save the values before exiting.

All values that have been changed, but not sent to the device, are handled at the termination of the method by either sending them to the device, discarding them, or saving the value changes in a way that survives the end of the method (but does not affect the device). Saved values can be used by other methods.

Communication profiles: FF, ISA100

If a method does not call builtin `discard_on_exit`, `save_on_exit`, or `send_on_exit` prior to exiting properly, any changes made by the method are discarded as if `discard_on_exit` was called.

Communication profiles: CS, GPE, PB, PN

If a method does not call builtin `discard_on_exit`, `save_on_exit`, or `send_on_exit` prior to exiting properly, any changes made by the method are saved as if `save_on_exit` was called.

If the method aborts, then the builtin `discard_on_exit` option is enforced even if builtin `save_on_exit` was called before the method aborted.

Lexical structure

`save_on_exit(VOID)`

The lexical element of the builtin `save_on_exit` is shown in Table 203.

Table 203 – Builtin `save_on_exit`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

`save_on_exit();`

6.6.3 Builtin `save_values`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Make VARIABLE values permanent

The builtin `save_values` immediately causes the values of all VARIABLES modified during a method session to remain (e.g. by committing them to the calling cache).

Communication profiles: HART

Any VARIABLES changed by the method are discarded when the method finishes running. These values are not automatically saved or sent to the device. Any VARIABLES modified after the builtin `save_values` was called will not remain unless the builtin `save_values` is called again. VARIABLES that are modified, but are not sent to the connected device, will be restored to their previous value if this function is not called.

Communication profiles: PB, PN, CS, GPE

`save_values` is used to save the VARIABLE values immediately. By default VARIABLE values are automatically saved when the method exits properly.

Lexical structure

`save_values(VOID)`

The lexical element of the builtin `save_values` is shown in Table 204.

Table 204 – Builtin `save_values`

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	One of the return codes specified in Table 377.

6.6.4 Builtin send_on_exit

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Send modified values to device

The builtin send_on_exit sets the termination action to send any changes made by the method to variables stored in the variable table of the EDD application, which have not already been sent to the device. This builtin does nothing in a non-caching EDD application, because a non-caching EDD application does not have a cache.

When a method modifies a variable value, it changes a copy of that value. This change is effective only for the life of the method. In order to make the change in the device, the method shall send the value(s) to the device or call this builtin before the method exits.

All values that are changed, but not sent to the device, are handled at the termination of the method by either sending them to the device, discarding them, or saving the value changes in a way that survives the end of the method. The changed values in the variable table are sent after the method terminates.

Communication profiles: FF, ISA100

If a method does not call builtin discard_on_exit, save_on_exit or send_on_exit prior to exiting properly, any changes made by the method are discarded as if discard_on_exit was called.

Communication profiles: CS, GPE, PB, PN

If a method does not call builtin discard_on_exit, save_on_exit or send_on_exit prior to exiting properly, any changes made by the method are saved as if save_on_exit was called.

If the method aborts then the builtin discard_on_exit option is enforced, even if builtin save_on_exit, or builtin send_on_exit were called before the method aborted.

Lexical structure

send_on_exit(VOID)

The lexical element of the builtin send_on_exit is shown in Table 205.

Table 205 – Builtin send_on_exit

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
send_on_exit();
```

6.7 Abnormal termination builtins

6.7.1 Builtin abort

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Abort the current method

The builtin abort will display a message indicating that the method has been aborted and wait for acknowledgement from the user. Once acknowledgement has been made, the system will execute any abort methods in the abort method list and will exit the method.

Lexical structure

`abort (VOID)`

The lexical element of the builtin abort is shown in Table 206.

Table 206 – Builtin abort

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.7.2 Builtin add_abort_method (version A)

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Add a method to the abort method list

The builtin `add_abort_method` adds a method to the bottom of the abort method list, which is the list of methods to be executed if the current method is aborted. The abort method list can hold up to 20 methods at any one time. The methods are executed in order starting from the top of the list and ending with the bottom. The same method may be added to the list more than once. A single instance of a method is removed from the abort method list after that method has completed execution.

Abort methods are only executed when the method is aborted, and not when the method exits under normal operating conditions. Methods can be aborted due to an abort mask condition when sending a command, or when the builtin abort or `process_abort` is called, or when the user aborts the method.

Lexical structure

`add_abort_method (abort_method_id)`

The lexical elements of the builtin `add_abort_method` are shown in Table 207.

Table 207 – Builtin add_abort_method

Parameter name	Type	Direction	Usage	Description
<code>abort_method</code>	reference	I	m	A reference to a METHOD.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the method was successfully added to the list, and BI_ERROR if the list was full.

6.7.3 Builtin add_abort_method (version B)

Communication profiles: FF, ISA100

Purpose: Add a method to the abort method list

The builtin add_abort_method adds an abort method to the calling method's abort method list. The abort method list is the list of methods which will be executed if the method aborts. The abort method list applies only to the current execution of the method. The abort method list can hold up to 20 methods at any one time.

Methods abort when response codes or communication errors occur which have been set to trigger an abort via other builtin calls, for example, abort_on_all_comm_errors and abort_on_all_response_codes. Methods can also be explicitly aborted by a direct call to builtin method_abort, or when an error occurs. The normal completion of a method is not considered an abort.

The abort method list is initially empty when a method begins execution. The abort methods are added to the back of the list and are removed and run from the front of the list in the order they appear, i.e. first-in first-out. An abort method may appear more than once in the list.

Abort methods may be shared between methods, meaning two methods may have the same abort method in their respective abort method lists.

The method_id parameter is specified by the method writer using the builtin ITEM_ID and the name of the method. Abort methods cannot call add_abort_method.

Lexical structure

add_abort_method(method_id)

The lexical elements of the builtin add_abort_method are shown in Table 208.

Table 208 – Builtin add_abort_method

Parameter name	Type	Direction	Usage	Description
method_id	unsigned long	I	m	The Item id of the method.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = add_abort_method(ITEM_ID(log_abort));
```

6.7.4 Builtin method_abort

Communication profiles: FF, ISA100

Purpose: Abort method and wait for acknowledge

The builtin method_abort displays a message indicating that the method is aborting and waits for acknowledgment from the user.

After the acknowledgment, the abort methods in the abort method list are executed one after another from the front of the list (where the first abort method was added to the list) to the back of the list. The abort method list is maintained only for the execution of a single invocation of the method.

After the abort methods are executed, the method terminates and control is returned to the application.

If an abort method calls `method_abort`, the abort method list processing is immediately terminated.

Lexical structure

```
method_abort(prompt)
```

The lexical elements of the builtin `method_abort` are shown in Table 209.

Table 209 – Builtin `method_abort`

Parameter name	Type	Direction	Usage	Description
prompt	DD_STRING	I	m	A message to be displayed (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

Example

```
method_abort("Method aborting...\n");
```

6.7.5 Builtin `pop_abort_method`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Removes an abort method from the top of the abort list

The builtin `pop_abort_method` removes an abort method from the top of the abort method list, which is the list of methods to be executed if the current method is aborted. Abort methods shall not be removed during an abort method.

Lexical structure

```
pop_abort_method(VOID)
```

The lexical element of the builtin `pop_abort_method` is shown in Table 210.

Table 210 – Builtin `pop_abort_method`

Parameter name	Type	Direction	Usage	Description
<return>	int	o	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the method was successfully removed from the list, and BI_ERROR if either no methods are in the list or if this function was run during an abort method.

6.7.6 Builtin `process_abort`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Run methods in the abort list

The builtin `process_abort` will abort the current method and execute any abort methods that are in the abort method list. Unlike the builtin `abort`, no message will be displayed when this function is executed. This builtin function may not be run from inside an abort method.

Lexical structure

`process_abort(VOID)`

The lexical element of the builtin `process_abort` is shown in Table 211.

Table 211 – Builtin `process_abort`

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

6.7.7 Builtin `push_abort_method`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Add a method to the top of the abort list

The builtin `push_abort_method` adds a method to the top of the abort method list, which is the list of methods to be executed if the current method is aborted. The abort method list can hold up to 20 methods at any one time. The methods are executed in order starting from the top of the list and ending with the bottom. The same method may be added to the list more than once. A single instance of a method is removed from the abort method list after that method has completed execution.

Abort methods are only executed when the method is aborted, and not when the method exits under normal operating conditions. Methods can be aborted due to an abort mask condition when sending a command, or when the builtin `abort` or `process_abort` is called, or when the user aborts the method.

Lexical Structure

`push_abort_method(abort_method_id)`

The lexical elements of the builtin `push_abort_method` are shown in Table 212.

Table 212 – Builtin `push_abort_method`

Parameter name	Type	Direction	Usage	Description
<code>abort_method</code>	reference	l	m	A reference to a METHOD.
<return>	int	o	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the method was successfully added to the list, and BI_ERROR if the list was full.

6.7.8 Builtin `remove_abort_method` (version A)

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Removes a method from the abort method list

The builtin `remove_abort_method` will remove a method from the abort method list, which is the list of methods to be executed if the current method is aborted. This builtin will remove the first occurrence of the specified method in the list, starting with the first method added. If there are multiple occurrences of a specific method, only the first one is removed. Abort methods shall not be removed during an abort method.

Lexical structure

`remove_abort_method(abort_method_name)`

The lexical elements of the builtin `remove_abort_method` are shown in Table 213.

Table 213 – Builtin remove_abort_method

Parameter name	Type	Direction	Usage	Description
abort_method	reference	I	m	A reference to a METHOD.
<return>	int	O	m	One of the return codes specified in Table 377.

6.7.9 Builtin remove_abort_method (version B)**Communication profiles: FF, ISA100****Purpose: Removes a method from the abort method list**

The builtin remove_abort_method moves a method from the abort method list. The abort method list contains the methods that are to be executed if the primary method aborts.

This builtin removes the first occurrence of the method that it finds in the list, starting at the front of the list and moving to the back. The front of the list has the abort methods that were added first (FIFO). If the method occurs more than once in the list, the subsequent occurrences are not removed.

The method_id parameter is specified by the method writer using the builtins ITEM_ID and the name of the method. Abort methods cannot call remove_abort_method.

Lexical structure

```
remove_abort_method(method_id)
```

The lexical elements of the builtin remove_abort_method are shown in Table 214.

Table 214 – Builtin remove_abort_method

Parameter name	Type	Direction	Usage	Description
method_id	unsigned long		m	The id of the method.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = remove_abort_method(ITEM_ID(log_info));
```

6.7.10 Builtin remove_all_abort_methods**Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN****Purpose: Remove all methods from abort list**

The builtin remove_all_abort_methods removes all methods from the abort method list. This builtin is similar to the builtin remove_abort_method (Version A) or remove_abort_method (Version B) except that it removes all of the abort methods. remove_all_abort_methods shall not be called from an abort method. If remove_all_abort_methods is encountered by the EDD application in an abort method, it shall be ignored.

Lexical structure

```
remove_all_abort_methods(VOID)
```

The lexical element of the builtin remove_all_abort_methods is shown in Table 215.

Table 215 – Builtin remove_all_abort_methods

Parameter name	Type	Direction	Usage	Description
<return>	VOID	—	—	—

Example

```
remove_all_abort_methods();
```

6.8 String builtins**6.8.1 Builtin atof**

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Convert string to floating-point

The builtin converts a string into a floating-point value.

Lexical structure

```
atof(str)
```

The lexical elements of the builtin atof are shown in Table 216.

Table 216 – Builtin atof

Parameter name	Type	Direction	Usage	Description
str	DD_STRING	I	m	The text to convert. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	double	O	m	The floating-point value produced by interpreting the input characters as a number. The return value is 0,0 if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

6.8.2 Builtin atoi

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Convert string to integer

The builtin converts a string into an integer value.

Lexical structure

```
atoi(str)
```

The lexical elements of the builtin atoi are shown in Table 217.

Table 217 – Builtin atoi

Parameter name	Type	Direction	Usage	Description
str	DD_STRING	I	m	The text to convert. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	The integer value produced by interpreting the input characters as a number. The return value 0 if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

6.8.3 Builtin BUILD_MESSAGE

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Build a string containing values of method-variables and/or VARIABLES

The builtin BUILD_MESSAGE returns a string based on a format specification. Any variable reference contained in the format string will be expanded to the current value. Method-variables and VARIABLES can be referenced in the message parameter. For more information, refer to the builtin PUT_MESSAGE.

Lexical structure

BUILD_MESSAGE(string)

The lexical elements of the builtin BUILD_MESSAGE are shown in Table 218.

Table 218 – Builtin BUILD_MESSAGE

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The format specification (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	The formatted string.

Example

```
int value;
DD_STRING message;
message = BUILD_MESSAGE("The value is %d.", value);
```

6.8.4 Builtin dictionary_string

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get dictionary string from name

The builtin dictionary_string returns the dictionary string associated with the given name in the current language. If the string is not available in the current language, the English string is returned. If the string is not found in the dictionary, the function returns "ERROR: Dict String Not Found". The parameter dict_string_name is the handle of the string as it appears in the dictionary.

NOTE Communication profiles HART: "ERROR: Dict String Not Found" is a standard string defined in the dictionary ([400,0] dict_str_not_found).

Lexical structure

`dictionary_string(dict_string_name)`

The lexical elements of the builtin `dictionary_string` are shown in Table 219.

Table 219 – Builtin `dictionary_string`

Parameter name	Type	Direction	Usage	Description
<code>dict_string_name</code>	reference	l	m	Handle of the string as it appears in the dictionary.
<return>	DD_STRING	o	m	The dictionary string if successful, "ERROR: Dict String Not Found" if string could not be found in the dictionary.

Example

```
ACKNOWLEDGE ( dictionary_string ( remove_loop_from_auto ) );
```

6.8.5 Builtin `ftoa`

Communication profiles: CS, GPE, PB, PN

Purpose: Convert float in string

The builtin converts a floating-point value into a text string.

Lexical structure

`ftoa(f)`

The lexical elements of the builtin `ftoa` are shown in Table 220.

Table 220 – Builtin `ftoa`

Parameter name	Type	Direction	Usage	Description
<code>f</code>	float	l	m	A floating-point value to convert to a text.
<return>	DD_STRING	o	m	The floating-point value as a string.

6.8.6 Builtin `get_comm_error_string`

Communication profiles: FF, ISA100

Purpose: Get string description for communication error code

The builtin `get_comm_error_string` retrieves the text description associated with a specific communications error code. If possible, the description of the communications error is used; if the description is not available, a default communications error message with the error number is used. The return code identifies which text description is returned.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary. A communications error of zero is not considered a communications error.

Lexical structure

`get_comm_error_string(error, string, maxlen)`

The lexical elements of the builtin `get_comm_error_string` are shown in Table 221.

Table 221 – Builtin get_comm_error_string

Parameter name	Type	Direction	Usage	Description
error	unsigned long	I	m	The communication error code.
string	DD_STRING	O	m	A description of the communication error. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
maxlen	long	I	m	The maximal length of string.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
char msg[256];
if (status == BLTIN_FAIL_COMM_ERROR)
{
    status = get_comm_error_string(get_comm_error(), msg, 256);
    status = get_acknowledgement(msg, 0, 0, 0,);
}

```

6.8.7 Builtin get_dictionary_string**Communication profiles: CS, GPE, HART, PB, PN****Purpose: Retrieve the text dictionary string**

The builtin get_dictionary_string will retrieve the text dictionary string associated with the given name in the current language. If the string is not available in the current language, the English string will be retrieved. If the string is not defined in either language, an error condition occurs, and the function will return BI_ERROR. If the string is longer than the max_str_len parameter, the string will be truncated. The parameter dict_string_name is the name of the string as it appears in the text dictionary.

Lexical structure

```
get_dictionary_string(dict_string_name, string, max_str_len)
```

The lexical elements of the builtin get_dictionary_string are shown in Table 222.

Table 222 – Builtin get_dictionary_string

Parameter name	Type	Direction	Usage	Description
dict_string_name	reference	I	m	The name of a dictionary entry.
string	DD_STRING	I	m	A buffer for the result string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
max_str_len	int	I	m	The maximum length of the buffer.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if successful, BI_ERROR if string could not be found.

Example

```
DD_STRING disp_string1; /* a variable to hold the string */
int slen1;
slen1 = 65535;          /* the maximum size to limit the string (use a
large value to get all of it) */

get_dictionary_string(remove_loop_from_auto, disp_string1, slen1);
ACKNOWLEDGE(disp_string1);
```

6.8.8 Builtin get_enum_string

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Get string for enumeration value

The builtin `get_enum_string` returns the description string associated of a specific enumeration element of an enumerated or bit enumerated variable. The enumeration element is specified by the value. The value shall be valid for the specified VARIABLE.

Lexical structure

```
get_enum_string(variable, value, string)
```

The lexical elements of the builtin `get_enum_string` are shown in Table 223.

Table 223 – Builtin get_enum_string

Parameter name	Type	Direction	Usage	Description
variable	DD_ITEM	I	m	An enumerated or bit enumerated variable.
value	int	I	m	A value of the variable.
string	DD_STRING	O	m	String associated with the value of the variable. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	One of the return codes specified in Table 376. BI_SUCCESS if successful, BI_ERROR if string could not be found.

6.8.9 Builtin get_response_code_string

Communication profiles: FF, ISA100

Purpose: Gets description of response code for specified item

The builtin `get_response_code_string` searches the response codes of the specified VARIABLE and returns the description of the specified response code. If the response code is not found, and if the item is a member of a RECORD or an element of a REFERENCE_ARRAY, then the RECORD or REFERENCE_ARRAY is checked for the specified code.

The method developer calls this builtin if the description of a response code is unknown and shall be displayed. Otherwise, if the response code is known, the method can determine what to do in the presence of certain errors. Also non-standard error descriptions can be displayed by an application instead of the standard response codes.

Before calling this builtin, the method shall use the builtin `get_response_code` to get the `item_id` and `member_id` for the item with a response code. The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID` and the name or reference of the VARIABLE.

Lexical structure

```
get_response_code_string(id, member_id, code, string, maxlen)
```

The lexical elements of the builtin `get_response_code_string` are shown in Table 224.

Table 224 – Builtin `get_response_code_string`

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the item to find.
member_id	unsigned long	I	m	The member id of the item to find.
code	unsigned long	I	m	The response code for which the descriptive string is to be found.
string	char*	O	m	The buffer for the returned response code description.
maxlen	long	I	m	The maximum length of the buffer.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long id;
unsigned long member;
unsigned long code;
long maxlen;
char msg[256];
if (status == BLTIN_FAIL_RESPONSE_CODE)
{
    maxlen = 256;
    status = get_response_code(&code, &id, &member);
    status = get_response_code_string(id, member, code, msg, maxlen);
    status = get_acknowledgement(msg, 0, 0, 0);
}
```

6.8.10 Builtin `get_rspcode_string`**Communication profiles: HART****Purpose: Get a response code string**

The builtin `get_rspcode_string` will return the response code string for the specified command and response code. If the string is longer than the maximum length defined in `length`, the string is truncated.

Lexical structure

```
get_rspcode_string(cmd_number, code, string, length)
```

The lexical elements of the builtin `get_rspcode_string` are shown in Table 225.

Table 225 – Builtin get_rspcode_string

Parameter name	Type	Direction	Usage	Description
cmd_number	int	I	m	The command number.
code	int	I	m	The response code.
string	DD_STRING	O	m	The buffer for the response code string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
length	int	I	m	The maximum length of the string buffer.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if successful, BI_ERROR if string could not be found.

6.8.11 Builtin get_rspcode_string_by_id**Communication profiles:** CS, GPE, PB, PN**Purpose:** Get a response code string

The builtin `get_rspcode_string_by_id` returns a string representation of a given `response_code` (returned by `ReadCommand`, `WriteCommand`).

Lexical structure

```
get_rspcode_string_by_id(command, response_code, result)
```

The lexical elements of the builtin `get_rspcode_string_by_id` are shown in Table 226.

Table 226 – Builtin get_rspcode_string_by_id

Parameter name	Type	Direction	Usage	Description
command	reference	I	m	The command from which the <code>response_code</code> was delivered.
response_code	int	I	m	The <code>response_code</code> value (as returned from <code>ReadCommand</code> or <code>WriteCommand</code>).
result	DD_STRING	O	m	String representation of the given <code>response_code</code> . String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	One of the return codes specified in Table 377.

6.8.12 Builtin get_status_code_string (deprecated)**Communication profiles:** HART**Purpose:** Get a status code string

The builtin `get_status_code_string` has the same functionality as builtin `get_enum_string`. If the string is longer than the maximum length defined in `status_string_length`, the string is truncated.

Lexical structure

```
get_status_code_string(variable_name, status_code, status_string,
status_string_length)
```

The lexical elements of the builtin `get_status_code_string` are shown in Table 227.

Table 227 – Builtin get_status_code_string

Parameter name	Type	Direction	Usage	Description
variable_name	char[]	I	m	The name of the VARIABLE to access.
status_code	int	I	m	The value of interesting status code.
status_string	DD_STRING	O	m	The buffer of the returned value of the signed VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
status_string_length	int	I	m	The maximum size of the buffer.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if successful, BI_ERROR if string could not be found.

6.8.13 Builtin get_status_string**Communication profiles: FF, ISA100****Purpose: Get description of a specific value of an enumerated or bit enumerated VARIABLE**

The builtin get_status_string searches the members of the referenced ENUMERATED or BIT_ENUMERATED VARIABLE and returns the description of the member matching the status in the given buffer.

The builtin gets the string associated with an ENUMERATED or BIT_ENUMERATED value. The most common use of ENUMERATED and BIT_ENUMERATED VARIABLES is to get a string associated with a status bit defined in the device for a particular status. The strings associated with ENUMERATED and BIT_ENUMERATED VARIABLES are defined in EDDL.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the variable in question.

Lexical structure

```
get_status_string(id, member_id, status, string, maxlen)
```

The lexical elements of the builtin get_status_string are shown in Table 228.

Table 228 – Builtin get_status_string

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the VARIABLE to access.
member_id	unsigned long	I	m	The member id of the VARIABLE to access.
status	unsigned long	I	m	The value to search.
string	char*	O	m	The buffer for the returned string.
maxlen	long	I	m	The maximum size of the buffer.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long device_status;
unsigned long id;
unsigned long member;
long maxlen;
char msg[256];
maxlen = 256;
id = ITEM_ID(PARAM.PV_SCALE);
member = MEMBER_ID(UNIT_CD);
status = get_status_string(id, member, device_status, msg, maxlen);

```

6.8.14 Builtin get_stdidict_string

Communication profiles: FF, ISA100

Purpose: Gets standard text dictionary string

The builtin `get_stdidict_string` takes a text dictionary identifier and returns the corresponding text string in the supplied string buffer. The returned string is in the correct language for the application.

A text dictionary entry is composed of three parts: a unique id, a name, and a multilingual description. The builtin strings are defined in the standard `dct` file.

Lexical structure

```
get_stdidict_string(id, string, maxlen)
```

The lexical elements of the builtin `get_stdidict_string` are shown in Table 229.

Table 229 – Builtin `get_stdidict_string`

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The text dictionary identifier.
string	char	O	m	The returned buffer for the string.
maxlen	long	I	m	The maximum size of the buffer.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
long len;
char msg[256];
len = 256;
status = get_stdidict_string(DICT_ID(square_root), msg, &len);

```

6.8.15 Builtin get_variable_string

Communication profiles: CS, GPE, PB, PN

Purpose: Get string representation of variable

The builtin `get_variable_string` will get a textual representation of the referenced VARIABLE.

Lexical structure

```
get_variable_string(device_var_name, destination_string)
```

The lexical elements of the builtin `get_variable_string` are shown in Table 230.

Table 230 – Builtin `get_variable_string`

Parameter name	Type	Direction	Usage	Description
device_var_name	reference	I	m	The name of the referenced VARIABLE.
destination_string	DD_STRING	O	m	Buffer for the textual representation of the referenced VARIABLE. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	One of the return codes specified in Table 377.

6.8.16 Builtin `itoa` (version A)

Communication profiles: CS, GPE, PB, PN

Purpose: Convert integer in string

The builtin converts an integer value into a string.

Lexical structure

```
itoa(i)
```

The lexical elements of the builtin `itoa` are shown in Table 231.

Table 231 – Builtin `itoa` (version A)

Parameter name	Type	Direction	Usage	Description
i	int	I	m	An integer value to convert to text.
<return>	DD_STRING	O	m	The returned text.

6.8.17 Builtin `itoa` (version B)

Communication profiles: HART

Purpose: Convert integer in string

The builtin `itoa` converts an integer value into a string. Radix is the base of the value (shall be between 2 and 36). If radix equals 10 and value is negative, the first character of the stored string shall be the minus sign (–).

Lexical structure

```
itoa(i, destination, radix)
```

The lexical elements of the builtin `itoa` are shown in Table 232.

Table 232 – Builtin itoa (version B)

Parameter name	Type	Direction	Usage	Description
i	int	I	m	An integer value to convert to text.
destination	DD_STRING	I	m	The string in which text is to be stored. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
radix	int	I	m	The base of the value (2 = binary, 8 = octal, 10 = decimal, 16 = hexadecimal).
<return>	DD_STRING	O	m	The returned string.

6.8.18 Builtin strleft

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Get substring from the beginning of a string

The builtin strleft extracts a substring from the beginning of a string.

Lexical structure

```
strleft(string, length)
```

The lexical elements of the builtin strleft are shown in Table 233.

Table 233 – Builtin strleft

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The input string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
length	int	I	m	The number of characters to be extracted from the input string.
<return>	DD_STRING	O	m	The specified substring.

6.8.19 Builtin strcmp

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Compares two strings

The builtin strcmp compares two strings lexicographically, and returns the relation as described in Table 234.

Lexical structure

```
strcmp(string1, string2)
```

The lexical elements of the builtin strcmp are shown in Table 234.

Table 234 – Builtin strcmp

Parameter name	Type	Direction	Usage	Description
string1	DD_STRING	I	m	The first string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
string2	DD_STRING	I	m	The second string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	Zero if string1 and string2 are the same, returns a negative value if string1 is less than string2, returns a positive value if string1 is greater than string2.

Example

```
DD_STRING buffer1 = "PT_101"
DD_STRING buffer2 = "PP_101"
int compare_result;
compare_result = strcmp( buffer1, buffer2 );
```

In this example, the value of compare_result will be greater than 0.

6.8.20 Builtin strlen

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Get the length of a string

The builtin strlen returns the length (number of characters) of a string.

Lexical structure

```
strlen(string)
```

The lexical elements of the builtin strlen are shown in Table 235.

Table 235 – Builtin strlen

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string whose length is to be returned. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	int	O	m	The length of the string.

Example

```
DD_STRING buffer;
buffer = "Unleash the power of EDDL!";
int len;
len = strlen(buffer);
```

In this example, the value of len will be set to 26.

6.8.21 Builtin strlwr

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Convert a string to lower case

The builtin strlwr returns a lower-case version of a string.

Lexical structure

strlwr(string)

The lexical elements of the builtin strlwr are shown in Table 236.

Table 236 – Builtin strlwr

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string to be converted. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	A lower-case version of the string.

Example

```
DD_STRING buffer;
long_tag = "Unleash the power of EDDL!";
send(22);
buffer = strlwr(long_tag);
```

In this example, buffer will be "unleash the power of eddl!".

6.8.22 Builtin strmid

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Get a substring from a string specifying starting position and length

The builtin strmid returns a substring of a string. It copies from the start position of string the amount of characters stated by length to a buffer and returns it. The first character of the string starts at position zero.

Lexical structure

strmid(string, position, length)

The lexical elements of the builtin strmid are shown in Table 237.

Table 237 – Builtin strmid

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The input string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
position	int	I	m	The first character to be extracted from the input string. This is a zero-based position (i.e. index 0 specifies the first character in the input string).
length	int	I	m	The number of characters to be extracted from the input string.
<return>	DD_STRING	O	m	The specified substring.

Example

```
DD_STRING string1 = "PT_101"
DD_STRING string2;
string2 = strmid(string1, 0, 2);
```

In this example, the value of string2 will be set to "PT" .

6.8.23 Builtin strright

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: get substring from the end of a string

The builtin strright extracts a substring from the end of a string.

Lexical structure

```
strright(string, length)
```

The lexical elements of the builtin strright are shown in Table 238.

Table 238 – Builtin strright

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The input string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
length	int	I	m	The number of characters to be extracted from the input string.
<return>	DD_STRING	O	m	The specified substring.

6.8.24 Builtin strstr

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Find a substring

The builtin strstr returns a substring of a string.

Lexical structure

```
strstr(string, search_string)
```

The lexical elements of the builtin strstr are shown in Table 239.

Table 239 – Builtin strstr

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string to be searched. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
search_string	DD_STRING	I	m	The search string. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	The substring beginning with the first occurrence of search_string, or an empty string ("") if the search_string cannot be found.

Example

```
DD_STRING buffer;
buffer = "Unleash the power of EDDL!";
tag = strstr(buffer, "power");
```

In this example, tag will contain the string "power of", as the call to strstr will return the DD_STRING "power of EDDL!", which will get truncated to 8 characters when converted to the tag ASCII DD-Variable.

6.8.25 Builtin strtrim

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Remove leading and trailing whitespace characters from a string

The builtin strtrim trims any leading and trailing whitespace from a string. Whitespace characters that are removed are space (' '), tab ('\t'), carriage return ('\r'), newline ('\n'), vertical tab ('\v'), and formfeed ('\f').

Lexical structure

```
strtrim(string)
```

The lexical elements of the builtin strtrim are shown in Table 240.

Table 240 – Builtin strtrim

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string to be trimmed. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	A string with leading and trailing whitespace removed.

Example

```
DD_STRING buffer1 = " PT_101 "
DD_STRING buffer2;
buffer2 = strtrim(buffer1);
```

In this example, the value of buffer2 will be set to "PT_101".

6.8.26 Builtinstrupr

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Convert a string to upper case

The builtinstrupr returns an upper-case version of a string.

Lexical structure

`strupr(string)`

The lexical elements of the builtinstrupr are shown in Table 241.

Table 241 – Builtinstrupr

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string to be converted. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	DD_STRING	O	m	An upper-case version of the string.

Example

```
DD_STRING buffer;
long_tag = "Unleash the power of EDDL!";
send(22);
buffer = strupr(long_tag);
```

In this example, buffer will be "UNLEASH THE POWER OF EDDL!", yet, long_tag will remain "Unleash the power of EDDL!".

6.8.27 Builtintrunc

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Truncate

The builtintrunc truncates a floating-point value to the nearest integer that is less than or equal to the specified floating point value.

Lexical structure

`trunc(x)`

The lexical elements of the builtintrunc are shown in Table 242.

Table 242 – Builtintrunc

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The truncated value rounded towards zero.

Example

```
double v, value;
v = trunc(value);
```

6.9 Date time builtins

6.9.1 Builtin AddTime

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Add time to a time value

The builtin AddTime adds a number of seconds to a time_t value.

Lexical structure

AddTime(t0, seconds)

The lexical elements of the builtin AddTime are shown in Table 243.

Table 243 – Builtin AddTime

Parameter name	Type	Direction	Usage	Description
t0	time_t	l	m	The initial time_t value.
seconds	double	l	m	The number of seconds to add.
<return>	time_t	o	m	the calculated time_t value.

Example

```
time_t t, tplus2;
tplus2 = AddTime(t0, 2.0);
```

6.9.2 Builtin DATE_AND_TIME_VALUE_to_string

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create formatted string from date and TIME_VALUE(4)

The builtin DATE_AND_TIME_VALUE_to_string creates a string representation of a DATE and a TIME_VALUE(4). The programming language-C strftime function defines the structure of the format string.

Lexical structure

DATE_AND_TIME_VALUE_to_string(string, format, date, time_value)

The lexical elements of the builtin DATE_AND_TIME_VALUE_to_string are shown in Table 244.

Table 244 – Builtin DATE_AND_TIME_VALUE_to_string

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I/O	m	The string into which the formatted DATE and TIME_VALUE are copied. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
format	DD_STRING	I	m	How the DATE and TIME_VALUE are to be formatted. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
date	long	I	m	A long value containing the DATE to be formatted.
time_value	unsigned long	I	m	The TIME_VALUE to be formatted.
<return>	int	O	m	The number of characters actually copied or -1 if an error occurs (e.g. format specifiers that are not applicable to the source data type).

6.9.3 Builtin Date_to_DayOfMonth**Communication profiles: CS, GPE, HART, PB, PN****Purpose: Calculate day of month from date**

The builtin Date_to_DayOfMonth calculates the day of the month from a given number of days since 1 January 1900.

Lexical structure

```
Date_to_DayOfMonth(date)
```

The lexical elements of the builtin Date_to_DayOfMonth are shown in Table 245.

Table 245 – Builtin Date_to_DayOfMonth

Parameter name	Type	Direction	Usage	Description
date	long	I	m	A long value containing the DATE.
<return>	int	O	m	The day of month. -1 is returned if the EDD application detects an invalid or erroneous DATE.

6.9.4 Builtin DATE_to_days**Communication profiles: CS, GPE, HART, PB, PN****Purpose: Calculate number of days between two dates**

The builtin DATE_to_days calculates the number of days between two dates.

Lexical structure

```
DATE_to_days(date1, date0)
```

The lexical elements of the builtin DATE_to_days are shown in Table 246.

Table 246 – Builtin DATE_to_days

Parameter name	Type	Direction	Usage	Description
date1	long	I	m	The more recent date.
date0	long	I	m	The less recent date.
<return>	long	O	m	The number of days between date1 and date0 (date1 – date0). If date1 is less than date 0, the result is unspecified.

6.9.5 Builtin Date_to_Month

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Calculate month from date

The builtin Date_to_Month calculates the month from a given number of days since 1 January 1900.

Lexical structure

Date_to_Month(date)

The lexical elements of the builtin Date_to_Month are shown in Table 247.

Table 247 – Builtin Date_to_Month

Parameter name	Type	Direction	Usage	Description
date	long	I	m	A long value containing the DATE.
<return>	int	O	m	The month. -1 is returned if the EDD application detects an invalid or erroneous DATE.

6.9.6 Builtin DATE_to_string

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create formatted string from date

The builtin DATE_to_string creates a string representation of a DATE. The programming language-C strftime function defines the structure of the format string.

Lexical structure

DATE_to_string(string, format, date)

The lexical elements of the builtin DATE_to_string are shown in Table 248.

Table 248 – Builtin DATE_to_string

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	I	m	The string into which the formatted DATE is copied. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
format	DD_STRING	I	m	How the DATE is to be formatted. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
date	long	I	m	A long value containing the DATE.
<return>	int	O	m	The number of character actually copied or -1 if an error occurs (e.g. format specifiers that are not applicable to the source data type).

6.9.7 Builtin Date_To_Time**Communication profiles:** CS, GPE, HART, PB, PN**Purpose:** Create time from date

The builtin Date_To_Time creates a time_t from a date value. The time of day is set to 00:00:00.

Lexical structure

Date_To_Time(date)

The lexical elements of the builtin Date_To_Time are shown in Table 249.

Table 249 – Builtin Date_To_Time

Parameter name	Type	Direction	Usage	Description
date	long	I	m	A long value containing the DATE.
<return>	time_t	O	m	A time_t value.

6.9.8 Builtin Date_to_Year**Communication profiles:** CS, GPE, HART, PB, PN**Purpose:** Calculate year from date

The builtin Date_to_Year calculates the year from a given number of days since 1 January 1900.

Lexical structure

Date_to_Year(date)

The lexical elements of the builtin Date_to_Year are shown in Table 250.

Table 250 – Builtin Date_to_Year

Parameter name	Type	Direction	Usage	Description
date	long	I	m	A long value containing the DATE.
<return>	int	O	m	The year (4 digits). -1 is returned if the EDD application detects an invalid or erroneous DATE.

6.9.9 Builtin days_to_DATE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Add days to date

The builtin days_to_DATE calculates a future date by adding a number of days to a DATE.

Lexical structure

days_to_DATE(days, date)

The lexical elements of the builtin days_to_DATE are shown in Table 251.

Table 251 – Builtin days_to_DATE

Parameter name	Type	Direction	Usage	Description
days	long	I	m	The number of days to add to the DATE.
date	long	I	m	A long value containing the initial DATE.
<return>	long	O	m	A long value containing the DATE.

6.9.10 Builtin DELAY_TIME

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Pause the current method for the number of seconds specified

The builtin DELAY_TIME will pause for the specified number of seconds. The delay time shall be a positive number. While DELAY_TIME is executing, the METHOD cannot be continued or aborted by the user.

Lexical structure

DELAY_TIME(delay_time)

The lexical elements of the builtin DELAY_TIME are shown in Table 252.

Table 252 – Builtin DELAY_TIME

Parameter name	Type	Direction	Usage	Description
delay_time	unsigned integer	I	m	The delay in number of seconds.
<return>	VOID	—	—	—

6.9.11 Builtin DiffTime

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate time difference

The builtin DiffTime calculates the difference between two calendar times and returns the difference in seconds as a double.

Lexical structure

DiffTime(t1, t0)

The lexical elements of the builtin DiffTime are shown in Table 253.

Table 253 – Builtin DiffTime

Parameter name	Type	Direction	Usage	Description
t1	time_t	l	m	The more recent time.
t0	time_t	l	m	The less recent time.
<return>	double	o	m	The difference between t1 and t0 in seconds.

Example

```
double seconds;
time_t t1, t0;
seconds = DiffTime(t1, t0);
```

6.9.12 Builtin From_DATE_AND_TIME_VALUE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create time from date and TIME_VALUE(4)

The builtin From_DATE_AND_TIME_VALUE creates a time_t from a DATE and a TIME_VALUE(4).

Lexical structure

From_DATE_AND_TIME_VALUE(date, time_value)

The lexical elements of the builtin From_DATE_AND_TIME_VALUE are shown in Table 254.

Table 254 – Builtin From_DATE_AND_TIME_VALUE

Parameter name	Type	Direction	Usage	Description
date	long	l	m	The DATE value.
time_value	unsigned long	l	m	The TIME_VALUE(4) value.
<return>	time_t	O	m	The time_t that was created or -1 if an invalid parameter is detected.

6.9.13 Builtin From_TIME_VALUE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create time from TIME_VALUE

The builtin From_TIME_VALUE creates a time_t from a TIME_VALUE(4) and the current date (midnight) or from a TIME_VALUE(8).

Lexical structure

From_TIME_VALUE(time_value)

The lexical elements of the builtin From_TIME_VALUE are shown in Table 255.

Table 255 – Builtin From_TIME_VALUE

Parameter name	Type	Direction	Usage	Description
time_value	unsigned long or unsigned long long	I	m	The TIME_VALUE value. A TIME_VALUE(8) is passed in as an unsigned long long and is an absolute time. A TIME_VALUE(4) is passed in as an unsigned long and is a time of day.
<return>	time_t	O	m	The time_t that was created or -1 if an invalid parameter is detected.

6.9.14 Builtin GetCurrentDate

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Return the current calendar date

The builtin GetCurrentDate returns the EDD application's best approximation of the current calendar date.

Lexical structure

```
GetCurrentDate()
```

The lexical element of the builtin GetCurrentDate is shown in Table 256.

Table 256 – Builtin GetCurrentDate

Parameter name	Type	Direction	Usage	Description
<return>	time_t	O	m	The current calendar date.

Example

```
time_t today;
today = GetCurrentDate();
```

6.9.15 Builtin GetCurrentDateAndTime

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Return the current calendar date and time

The builtin GetCurrentDateAndTime returns the EDD application's best approximation of the current calendar date and time.

Lexical structure

```
GetCurrentDateAndTime()
```

The lexical element of the builtin GetCurrentDateAndTime is shown in Table 257.

Table 257 – Builtin GetCurrentDateAndTime

Parameter name	Type	Direction	Usage	Description
<return>	time_t	o	m	The current calendar date and time.

Example

```
time_t today;
today = GetCurrentDateAndTime();
```

6.9.16 Builtin GetCurrentTime

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Get current EDD application time

The builtin GetCurrentTime gets the current time from the EDD application's best approximation of the current calendar time.

Lexical structure

GetCurrentTime(VOID)

The lexical element of the builtin GetCurrentTime is shown in Table 258.

Table 258 – Builtin GetCurrentTime

Parameter name	Type	Direction	Usage	Description
<return>	time_t	o	m	The current calendar time.

Example

```
time_t today;
today = GetCurrentTime();
```

6.9.17 Builtin GET_TICK_COUNT

Communication profiles: CS, GPE, PB, PN

Purpose: Get time in ms

The builtin GET_TICK_COUNT returns the time in ms since the last system boot. It can be used for time stamps.

Lexical structure

GET_TICK_COUNT(VOID)

The lexical element of the builtin GET_TICK_COUNT is shown in Table 259.

Table 259 – Builtin GET_TICK_COUNT

Parameter name	Type	Direction	Usage	Description
<return>	long	O	m	The time in ms since the last system boot.

6.9.18 Builtin Make_Time

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Create Time from input parameters

The builtin Make_Time creates a time_t from its input parameters. It allows an arbitrary calendar time to be created. The year is the calendar year (e.g., 2004) and the hour is in the range 0 to 23. The isDST is non-zero if the time is the local daylight savings time (i.e., summer-time).

Lexical structure

`Make_Time(year, month, day, hour, minute, second, isDST)`

The lexical elements of the builtin `Make_Time` are shown in Table 260.

Table 260 – Builtin `Make_Time`

Parameter name	Type	Direction	Usage	Description
year	int	I	m	The year.
month	int	I	m	The month.
day	int	I	m	The day.
hour	int	I	m	The hour.
minute	int	I	m	The minute.
second	int	I	m	The second.
isDST	int	I	m	Specifies whether daylight savings time is in effect. If zero, daylight savings time is not in effect; otherwise, daylight savings time is in effect.
<return>	time_t	O	m	The time_t that was created or -1 if an invalid parameter is detected.

Example

```
time_t t;
t = Make_Time(1,1,1970, 0, 0, 0, 0);
```

6.9.19 Builtin `seconds_to_TIME_VALUE`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Convert seconds to `TIME_VALUE(4)`

The builtin `seconds_to_TIME_VALUE` converts a seconds value to a `TIME_VALUE`.

Lexical structure

`seconds_to_TIME_VALUE(seconds)`

The lexical element of the builtin `seconds_to_TIME_VALUE` is shown in Table 261.

Table 261 – Builtin `seconds_to_TIME_VALUE`

Parameter name	Type	Direction	Usage	Description
seconds	double	I	m	The number of seconds.
<return>	unsigned long	O	m	The <code>TIME_VALUE(4)</code> value.

Example

```
unsigned long v;
double secs;
v = seconds_to_TIME_VALUE(secs);
```

6.9.20 Builtin seconds_to_TIME_VALUE8**Communication profiles:** CS, FF, GPE, ISA100, PB, PN**Purpose:** Convert seconds to a TIME_VALUE(8)

The builtin seconds_to_TIME_VALUE8 converts a seconds value to a TIME_VALUE(8).

Lexical structure

seconds_to_TIME_VALUE8(seconds)

The lexical element of the builtin seconds_to_TIME_VALUE is shown in Table 262.

Table 262 – Builtin seconds_to_TIME_VALUE

Parameter name	Type	Direction	Usage	Description
seconds	double	I	m	The number of seconds.
<return>	unsigned long long	O	m	The TIME_VALUE(8) value.

6.9.21 Builtin Time_To_Date**Communication profiles:** CS, GPE, HART, PB, PN**Purpose:** Create date from time

The builtin Time_To_Date creates a DATE value from a calendar date and time.

Lexical structure

Time_To_Date(t)

The lexical elements of the builtin Time_To_Date are shown in Table 263.

Table 263 – Builtin Time_To_Date

Parameter name	Type	Direction	Usage	Description
t	time_t	I	m	The time_t value.
<return>	long	O	m	The DATE value.

6.9.22 Builtin TIME_VALUE_to_Hour**Communication profiles:** CS, FF, GPE, HART, ISA100, PB, PN**Purpose:** Extract hour from TIME_VALUE

The builtin TIME_VALUE_to_Hour extracts the hour component from a TIME_VALUE.

Lexical structure

TIME_VALUE_to_Hour(time_value)

The lexical elements of the builtin TIME_VALUE_to_Hour are shown in Table 264.

Table 264 – Builtin TIME_VALUE_to_Hour

Parameter name	Type	Direction	Usage	Description
time_value	unsigned long or unsigned long long	I	m	The TIME_VALUE. A TIME_VALUE(8) is passed in as an unsigned long long and is an absolute time. A TIME_VALUE(4) is passed in as an unsigned long and is a duration or a time of day.
<return>	int	O	m	The hour component.

Example

```
int hour;
unsigned long value;
hour = TIME_VALUE_to_Hour(value);
```

6.9.23 Builtin TIME_VALUE_to_Minute

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Extract minute from TIME_VALUE

The builtin TIME_VALUE_to_Minute extracts the minute component from a TIME_VALUE.

Lexical structure

```
TIME_VALUE_to_Minute(time_value)
```

The lexical elements of the builtin TIME_VALUE_to_Minute are shown in Table 265.

Table 265 – Builtin TIME_VALUE_to_Minute

Parameter name	Type	Direction	Usage	Description
time_value	unsigned long or unsigned long long	I	m	The TIME_VALUE. A TIME_VALUE(8) is passed in as an unsigned long long and is an absolute time. A TIME_VALUE(4) is passed in as an unsigned long and is a duration or a time of day.
<return>	int	O	m	The minute component.

Example

```
int minute;
unsigned long value;
minute = TIME_VALUE_to_Minute(value);
```

6.9.24 Builtin TIME_VALUE_to_Second

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Extract second from TIME_VALUE

The builtin TIME_VALUE_to_Second extracts the seconds component from a TIME_VALUE.

Lexical structure

```
TIME_VALUE_to_Second(time_value)
```

The lexical elements of the builtin TIME_VALUE_to_Second are shown in Table 266.

Table 266 – Builtin TIME_VALUE_to_Second

Parameter name	Type	Direction	Usage	Description
time_value	unsigned long or unsigned long long	I	m	The TIME_VALUE. A TIME_VALUE(8) is passed in as an unsigned long long and is an absolute time. A TIME_VALUE(4) is passed in as an unsigned long and is a duration or a time of day.
<return>	int	O	m	The seconds component.

Example

```
int second;
unsigned long value;
second = TIME_VALUE_to_Second(value);
```

6.9.25 Builtin TIME_VALUE_to_seconds

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Convert TIME_VALUE to number of seconds

The builtin TIME_VALUE_to_seconds converts a TIME_VALUE to a number of seconds.

Lexical structure

```
TIME_VALUE_to_seconds(time_value)
```

The lexical elements of the builtin TIME_VALUE_to_seconds are shown in Table 267.

Table 267 – Builtin TIME_VALUE_to_seconds

Parameter name	Type	Direction	Usage	Description
time_value	unsigned long or unsigned long long	I	m	The TIME_VALUE. A TIME_VALUE(8) is passed in as an unsigned long long and is an absolute time. A TIME_VALUE(4) is passed in as an unsigned long and is a duration or a time of day.
<return>	double	O	m	The number of seconds of the duration or seconds since midnight if a TIME_VALUE(4) was given or the number of seconds since January 1 st 1972 if a TIME_VALUE(8) was given.

Example

```
double seconds;
unsigned long value;
seconds = TIME_VALUE_to_seconds(value);
```

6.9.26 Builtin TIME_VALUE_to_string

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Create formatted string from TIME_VALUE

The builtin TIME_VALUE_to_string creates a string representation of a TIME_VALUE. The programming language-C strftime function defines the structure of the format string.

Lexical structure

`TIME_VALUE_to_string(string, format, time_value)`

The lexical elements of the builtin `TIME_VALUE_to_string` are shown in Table 268.

Table 268 – Builtin `TIME_VALUE_to_string`

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	O	m	The string into which the formatted <code>TIME_VALUE</code> is copied. String types that are not a <code>DD_STRING</code> may be also be used and will be automatically promoted to a <code>DD_STRING</code> (see 6.18.3).
format	DD_STRING	I	m	How the <code>TIME_VALUE</code> is to be formatted. String types that are not a <code>DD_STRING</code> may be also be used and will be automatically promoted to a <code>DD_STRING</code> (see 6.18.3).
time_value	unsigned long or unsigned long long	I	m	The <code>TIME_VALUE</code> to be formatted. A <code>TIME_VALUE(8)</code> is passed in as an unsigned long long and is an absolute time. A <code>TIME_VALUE(4)</code> is passed in as an unsigned long and is a duration or a time of day.
<return>	int	O	m	The number of characters actually copied or -1 if an error occurs (e.g. format specifiers that are not applicable to the source data type).

Example

```
DD_STRING str[50];
int length;
unsigned long value;
length = TIME_VALUE_to_string(str, "%m/%d/%y", value);
```

6.9.27 Builtin `timet_to_string`

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Create a formatted string from time

The builtin `timet_to_string` creates a string representation of a `time_t`. The programming language-C `strftime` function defines the structure of the format string.

Lexical structure

`timet_to_string(string, format, timet)`

The lexical elements of the builtin `timet_to_string` are shown in Table 269.

Table 269 – Builtin `timet_to_string`

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	O	m	The string into which formatted <code>time_t</code> is copied. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
format	DD_STRING	I	m	How the <code>time_t</code> is to be formatted. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
timet	time_t	I	m	The <code>time_t</code> to be formatted.
<return>	int	O	m	The number of characters actually copied or -1 if an error occurs (e.g. format specifiers that are not applicable to the source data type).

Example

```
DD_STRING str[50];
int length;
time_t value;
length = timet_to_string(str, "%m/%d/%y", value);
```

6.9.28 Builtin `timet_to_TIME_VALUE`**Communication profiles:** CS, GPE, PB, PN**Purpose:** Convert time to TIME_VALUE(4)

The builtin `timet_to_TIME_VALUE` converts the time of day part of a `time_t` to a TIME_VALUE(4).

Lexical structure

```
timet_to_TIME_VALUE(timet)
```

The lexical elements of the builtin `timet_to_TIME_VALUE` are shown in Table 270.

Table 270 – Builtin `timet_to_TIME_VALUE`

Parameter name	Type	Direction	Usage	Description
timet	time_t	I	m	The <code>time_t</code> value to convert.
<return>	unsigned long	O	m	The TIME_VALUE(4) value corresponding to the time of day given by the input parameter <code>timet</code> .

Example

```
unsigned long t;
time_t value;
t = timet_to_TIME_VALUE(value);
```

6.9.29 Builtin `timet_To_TIME_VALUE`**Communication profiles:** HART**Purpose:** Convert time to TIME_VALUE(4)

The builtin `timet_To_TIME_VALUE` converts the time of day part of a `time_t` to a TIME_VALUE(4).

Lexical structure

```
timet_To_TIME_VALUE(timet)
```

The lexical elements of the builtin `timet_To_TIME_VALUE` are shown in Table 271.

Table 271 – Builtin `timet_To_TIME_VALUE`

Parameter name	Type	Direction	Usage	Description
timet	time_t	I	m	The time_t value to convert.
<return>	unsigned long	O	m	The TIME_VALUE(4) value corresponding to the time of day given by the input parameter timet.

6.9.30 Builtin `timet_to_TIME_VALUE8`

Communication: Profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Convert time to TIME_VALUE(8)

The builtin `timet_to_TIME_VALUE` converts time_t to a TIME_VALUE(8).

Lexical structure

```
timet_to_TIME_VALUE8(timet)
```

The lexical elements of the builtin `timet_to_TIME_VALUE` are shown in Table 272.

Table 272 – Builtin `timet_to_TIME_VALUE8`

Parameter name	Type	Direction	Usage	Description
timet	time_t	I	m	The time_t value to convert.
<return>	unsigned long long	O	m	The TIME_VALUE(8) value corresponding to the date and time given by the input parameter timet.

6.9.31 Builtin `To_Date`

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create date from input parameters

The builtin `To_Date` creates a DATE from year, month and day. The year is the calendar year (e.g., 2004). The EDD application may verify the validity of the parameters and set the result to the nearest legal DATE.

Lexical structure

```
To_Date(year, month, day)
```

The lexical elements of the builtin `To_Date` are shown in Table 273.

Table 273 – Builtin `To_Date`

Parameter name	Type	Direction	Usage	Description
year	int	I	m	The year.
month	int	I	m	The month.
day	int	I	m	The day.
<return>	long	O	m	The DATE value.

6.9.32 Builtin To_Date_and_Time**Communication profiles: CS, GPE, PB, PN****Purpose: Create DATE_AND_TIME from input parameters**

The builtin To_Date_and_Time converts a days, hour, minute, second and millisecond value to a DATE_AND_TIME value.

NOTE The return value can be assigned to a DATE_AND_TIME variable.

Lexical structure

To_Date_and_Time(days, hour, minute, second, millisecond)

The lexical elements of the builtin To_Date_and_Time are shown in Table 274.

Table 274 – Builtin To_Date_and_Time

Parameter name	Type	Direction	Usage	Description
days	int	I	m	The number of days since January 1 st , 1900. The number of days can only be a positive number.
hour	int	I	m	The hour.
minute	int	I	m	The minute.
second	int	I	m	The second.
millisecond	int	I	m	The millisecond.
<return>	unsigned long long	O	m	The DATE_AND_TIME value corresponding to the given time_t value packaged in an unsigned long long value.

6.9.33 Builtin To_Time**Communication profiles: CS, GPE, HART, PB, PN****Purpose: Create a time from input parameters**

The builtin To_Time creates a time_t value from its input arguments.

Lexical structure

To_Time(date, hour, minute, second, isDST)

The lexical elements of the builtin To_Time are shown in Table 275.

Table 275 – Builtin To_Time

Parameter name	Type	Direction	Usage	Description
date	long	I	m	A long value containing the DATE.
hour	int	I	m	The hour.
minute	int	I	m	The minute.
second	int	I	m	The second.
isDST	int	I	m	Specifies whether daylight savings time is in effect. If zero, daylight savings time is not in effect; otherwise, daylight savings time is in effect.
<return>	time_t	O	m	The time_t value.

6.9.34 Builtin To_TIME_VALUE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Create TIME_VALUE(4) from input parameters

The builtin To_TIME_VALUE creates a TIME_VALUE from its input arguments.

Lexical structure

To_TIME_VALUE(hours, minutes, seconds)

The lexical elements of the builtin To_TIME_VALUE are shown in Table 276.

Table 276 – Builtin To_TIME_VALUE

Parameter name	Type	Direction	Usage	Description
hours	int	I	m	The number of hours.
minutes	int	I	m	The number of minutes.
seconds	int	I	m	The number of seconds.
<return>	unsigned long	O	m	The TIME_VALUE(4) value corresponding to the time of day given by the input parameter hours, minutes and seconds.

Example

```
double time;
time = To_TIME_VALUE(12, 0, 0);
```

6.9.35 Builtin To_TIME_VALUE8

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Create TIME_VALUE(8) from input parameters

The builtin To_TIME_VALUE8 creates a TIME_VALUE(8) from its input arguments.

Lexical structure

To_TIME_VALUE8(year, month, day, hour, minute, second)

The lexical elements of the builtin To_TIME_VALUE are shown in Table 277.

Table 277 – Builtin To_TIME_VALUE8

Parameter name	Type	Direction	Usage	Description
year	int	I	m	The year.
month	int	I	m	The month.
day	int	I	m	The day.
hour	int	I	m	The hour.
minute	int	I	m	The minute.
second	int	I	m	The second.
<return>	unsigned long long	O	m	The TIME_VALUE(8) value corresponding to the input parameters.

6.10 Diagnostic builtins

6.10.1 Builtin `get_comm_error`

Communication profiles: FF, ISA100

Purpose: Get communication error code

The builtin `get_comm_error` returns the communications error code received from an application in the event of a builtin return code of `BLTIN_FAIL_COMM_ERROR`.

This builtin is used primarily for debugging purposes to determine what type of communications failures are occurring by displaying the communications error for the method writer to see.

In rare cases, the method could react to specific communications errors. However, the method should specify alternate handling of communications errors by way of the following:

- `fail_on_all_comm_errors`
- `fail_on_comm_error`
- `abort_on_all_comm_errors`
- `ABORT_ON_COMM_ERROR`

The alternate handling may cause the method to abort or immediately process the error based on the presence of a communications error.

The communications error codes are defined by the consortia. Their related strings may be defined in a text dictionary. A communications error of zero is not considered a communications error. The builtin gets the communications error code returned from the last communications request.

Lexical structure

`get_comm_error(VOID)`

The lexical element of the builtin `get_comm_error` is shown in Table 278.

Table 278 – Builtin `get_comm_error`

Parameter name	Type	Direction	Usage	Description
<return>	unsigned long	O	m	A communication error code.

Example

```
long status;
if (status == BLTIN_FAIL_COMM_ERROR)
{
    status = display_comm_error(get_comm_error());
}
```

6.10.2 Builtin `get_dds_error`

Communication profiles: FF, ISA100

Purpose: Return EDD error code

The builtin `get_dds_error` returns the EDD error generated by an earlier call, and optionally returns a text string describing the error.

By calling this builtin, a method may determine the specific EDD error code related to the received return code BLTIN_DDS_ERROR. A textual representation of the error message is put into a provided buffer with a given maximum length.

This builtin provides a means of debugging calls to builtin routines, which are returning an error. It should never be used in a final production method, because the types of errors that are returned with get_dds_error should never be received in a fully debugged method.

The EDD error and optional description returned are for the most recent EDD error.

Lexical structure

get_dds_error(string, maxlen)

The lexical elements of the builtin get_dds_error are shown in Table 279.

Table 279 – Builtin get_dds_error

Parameter name	Type	Direction	Usage	Description
string	DD_STRING	O	m	An output buffer for the description of the error. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
maxlen	long	I	m	The maximum length of the output buffer (string) in octets.
<return>	long	O	m	An error code generated by an earlier builtin call.

Example

```
long status;
char msg[256];
if(status == BLTIN_DDS_ERROR)
{
    status = get_dds_error(msg, 256);
}
```

6.10.3 Builtin get_response_code

Communication profiles: FF, ISA100

Purpose: Get response code for last communication request

The builtin get_response_code returns the response code received from a device in the event of a builtin communications call returning the error BLTIN_FAIL_RESPONSE_CODE. It also returns the value reference (err_id, err_member_id) information associated with the original request. The original request is the last builtin call that caused communication.

A response code is a value representing an application-specific error condition.

NOTE A response code value of zero is not considered an error response code. Each response code and response code type is defined in the EDD.

Gets the response code returned on the last communications request, and the item id and member id for the value being accessed when the response code was returned.

Lexical structure

get_response_code(resp_code, err_id, err_member_id)

The lexical elements of the builtin get_response_code are shown in Table 280.

Table 280 – Builtin get_response_code

Parameter name	Type	Direction	Usage	Description
resp_code	unsigned long	O	m	The number of the returned response code.
err_id	unsigned long	O	m	The returned item id of the item generating the response code.
err_member_id	unsigned long	O	m	The returned member id of the item causing a response code.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long id;
unsigned long member;
unsigned long code;
if (status == BLTIN_FAIL_RESPONSE_CODE)
{
    status = get_response_code(&code, &id, &member);
    status = display_response_code(id, member, code);
}

```

6.11 Math builtins**6.11.1 Builtin abs**

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate absolute value

The builtin abs returns the absolute value of a floating-point value.

Lexical structure

abs(x)

The lexical elements of the builtin abs are shown in Table 281.

Table 281 – Builtin abs

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The absolute value.

Example

```

double v, value;
v = abs(value);

```

6.11.2 Builtin acos

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate arc cosine

The builtin acos returns the arc cosine of a floating-point value. The value is forced to be in the range [-1, +1].

Lexical structure

`acos(x)`

The lexical elements of the builtin `acos` are shown in Table 282.

Table 282 – Builtin `acos`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The arc cosine.

Example

```
double v, value;
v = acos(value);
```

6.11.3 Builtin `asin`

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate arc sine

The builtin `asin` returns the arc sine of a floating-point value. The value is forced to be in the range $[-1, +1]$.

Lexical structure

`asin(x)`

The lexical elements of the builtin `asin` are shown in Table 283.

Table 283 – Builtin `asin`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The arc sine.

Example

```
double v, value;
v = asin(value);
```

6.11.4 Builtin `atan`

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate arc tangent

The builtin `atan` returns the arc tangent of a floating-point value. The value is forced to be in the range $[-1, +1]$.

Lexical structure

`atan(x)`

The lexical elements of the builtin `atan` are shown in Table 284.

Table 284 – Builtin atan

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The arc tangent.

Example

```
double v, value;
v = atan(value);
```

6.11.5 Builtin ByteToDouble

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate double from bytes

The builtin ByteToDouble will return a double value from a vector of eight octets.

Lexical structure

```
ByteToDouble(in1, in2, in3, in4, in5, in6, in7, in8)
```

The lexical element of the builtin ByteToDouble is shown in Table 285.

Table 285 – Builtin ByteToDouble

Parameter name	Type	Direction	Usage	Description
in1	unsigned char	I	m	First element of the double value (most significant byte).
in2	unsigned char	I	m	Second element of the double value.
in3	unsigned char	I	m	Third element of the double value.
in4	unsigned char	I	m	Fourth element of the double value.
in5	unsigned char	I	m	Fifth element of the double value.
in6	unsigned char	I	m	Sixth element of the double value.
in7	unsigned char	I	m	Seventh element of the double value.
in8	unsigned char	I	m	Eighth element of the double value (least significant byte).
<return>	double	O	m	A double value from the given vector of eight octets.

6.11.6 Builtin ByteToFloat

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate float from bytes

The builtin ByteToFloat will return a floating value from a vector of four octets.

Lexical structure

```
ByteToFloat(in1, in2, in3, in4)
```

The lexical elements of the builtin ByteToFloat are shown in Table 286.

Table 286 – Builtin ByteToFloat

Parameter name	Type	Direction	Usage	Description
in1	unsigned char	I	m	First element of the float value (most significant byte).
in2	unsigned char	I	m	Second element of the float value.
in3	unsigned char	I	m	Third element of the float value.
in4	unsigned char	I	m	Fourth element of the float value (least significant byte).
<return>	float	O	m	A floating-point value from the given vector of four octets.

6.11.7 Builtin ByteToLong

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate long from bytes

The builtin ByteToLong will return a long value from a vector of four octets.

Lexical structure

ByteToLong(in1, in2, in3, in4)

The lexical elements of the builtin ByteToLong are shown in Table 287.

Table 287 – Builtin ByteToLong

Parameter name	Type	Direction	Usage	Description
in1	unsigned char	I	m	First element of the long value (most significant byte).
in2	unsigned char	I	m	Second element of the long value.
in3	unsigned char	I	m	Third element of the long value.
in4	unsigned char	I	m	Fourth element of the long value (least significant byte).
<return>	long	O	m	A long value from the given vector of four octets.

6.11.8 Builtin ByteToShort

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate short from bytes

The builtin ByteToShort will return a short value from a vector of two octets.

Lexical structure

ByteToShort(in1, in2)

The lexical elements of the builtin ByteToShort are shown in Table 288.

Table 288 – Builtin ByteToShort

Parameter name	Type	Direction	Usage	Description
in1	unsigned char	I	m	First element of the short value (most significant byte).
in2	unsigned char	I	m	Second element of the short value (least significant byte).
<return>	short	O	m	A short value from the given vector of two octets.

6.11.9 Builtin cbrt

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate cubic root

The builtin cbrt returns the cubic root of a floating-point value.

Lexical structure

`cbrt(x)`

The lexical elements of the builtin cbrt are shown in Table 289.

Table 289 – Builtin cbrt

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The cubic root.

Example

```
double v, value;
v = cbrt(value);
```

6.11.10 Builtin ceil

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate next higher integer

The builtin ceil returns the smallest integral value not less than a specified floating-point value (next higher integer value).

Lexical structure

`ceil(x)`

The lexical elements of the builtin ceil are shown in Table 290.

Table 290 – Builtin ceil

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The ceiling value.

Example

```
double v, value;
v = ceil(value);
```

6.11.11 Builtin cos

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate cosine

The builtin cos returns the cosine of a floating-point value (measured in radians).

Lexical structure

cos(x)

The lexical elements of the builtin cos are shown in Table 291.

Table 291 – Builtin cos

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The cosine.

Example

```
double v, value;
v = cos(value);
```

6.11.12 Builtin cosh

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate hyperbolic cosine

The builtin cosh returns the hyperbolic cosine of a floating-point value.

Lexical structure

cosh(x)

The lexical elements of the builtin cosh are shown in Table 292.

Table 292 – Builtin cosh

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The hyperbolic cosine.

Example

```
double v, value;
v = cosh(value);
```

6.11.13 Builtin DoubleToByte**Communication profiles:** CS, GPE, PB, PN**Purpose:** Calculate Bytes from double

The builtin DoubleToByte will provide a vector of unsigned char values representing the particular elements of a double value.

Lexical structure

```
DoubleToByte(in, out1, out2, out3, out4, out5, out6, out7, out8)
```

The lexical elements of the builtin DoubleToByte are shown in Table 293.

Table 293 – Builtin DoubleToByte

Parameter name	Type	Direction	Usage	Description
in	double	l	m	The double value to convert.
out1	unsigned char	O	m	First element of the double value (most significant byte).
out2	unsigned char	O	m	Second element of the double value.
out3	unsigned char	O	m	Third element of the double value.
out4	unsigned char	O	m	Fourth element of the double value.
out5	unsigned char	O	m	Fifth element of the double value.
out6	unsigned char	O	m	Sixth element of the double value.
out7	unsigned char	O	m	Seventh element of the double value.
out8	unsigned char	O	m	Eighth element of the double value (least significant byte).
<return>	VOID	—	—	—

6.11.14 Builtin drand**Communication profiles:** CS, FF, GPE, HART, ISA100, PB, PN**Purpose:** Calculate random value

The builtin drand() returns random double precision values uniformly distributed across the range from 0 to 1.

Lexical structure

```
double drand()
```

The lexical element of the builtin drand is shown in Table 294.

Table 294 – Builtin drand

Parameter name	Type	Direction	Usage	Description
<return>	double	O	m	Returns random double precision values uniformly distributed across the range from 0 to 1.

Example

```
double v;
v = drand();
```


6.11.15 Builtin dseed

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate seed based random value

The builtin dseed() takes a double in the range 0 to 1 as the seed for the pseudo-random number generator. For a given EDD application implementation, if dseed is called with the same seed value the same pseudo-random number sequence shall be generated.

Lexical structure

```
void dseed(double seed)
```

The lexical elements of the builtin dseed are shown in Table 295.

Table 295 – Builtin dseed

Parameter name	Type	Direction	Usage	Description
seed	double	I	m	A double in the range 0 to 1 as the seed for the pseudo-random number generator.
<return>	VOID	—	—	—

Example

```
dseed(0.452132);
```

6.11.16 Builtin exp

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate exponential

The builtin exp returns the exponential of a floating-point value.

Lexical structure

```
exp(x)
```

The lexical elements of the builtin exp are shown in Table 296.

Table 296 – Builtin exp

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The exponential value.

Example

```
double v, value;
v = exp(value);
```

6.11.17 Builtin FloatToByte

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate bytes from float

The builtin FloatToByte will provide a vector of unsigned char values representing the particular elements of a float value.

Lexical structure

```
FloatToByte(in, out1, out2, out3, out4)
```

The lexical element of the builtin FloatToByte is shown in Table 297.

Table 297 – Builtin FloatToByte

Parameter name	Type	Direction	Usage	Description
in	float	I	m	The float value to convert.
out1	unsigned char	O	m	First element of the float value (most significant byte).
out2	unsigned char	O	m	Second element of the float value.
out3	unsigned char	O	m	Third element of the float value.
out4	unsigned char	O	m	Fourth element of the float value (least significant byte).
<return>	VOID	—	—	—

6.11.18 Builtin floor

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate next lower integer

The builtin floor returns the largest integral value not greater than a specified floating-point value (next lower integer).

Lexical structure

```
floor(x)
```

The lexical elements of the builtin floor are shown in Table 298.

Table 298 – Builtin floor

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The floor value.

Example

```
double v, value;
v = floor(value);
```

6.11.19 Builtin fmod

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate remainder

The builtin fmod returns the floating-point remainder of the division of two floating-point values.

Lexical structure

```
fmod(x, y)
```

The lexical elements of the builtin fmod are shown in Table 299.

Table 299 – Builtin fmod

Parameter name	Type	Direction	Usage	Description
x	double	I	m	The numerator.
y	double	I	m	The denominator.
<return>	double	O	m	The remainder.

Example

```
double r, n, d;
r = fmod(n, d);
```

6.11.20 Builtin fpclassify

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Provides information about a floating point number

The builtin fpclassify classifies a floating-point value as either not a number, infinite, zero, subnormal, or normal.

NOTE This builtin is implemented as a macro that generates a call to either fpclassifyf or fpclassifyd depending on whether its input parameter is a float or double, respectively.

Lexical structure

```
fpclassify(value)
```

The lexical elements of the builtin fpclassify are shown in Table 300.

Table 300 – Builtin fpclassify

Parameter name	Type	Direction	Usage	Description
value	float or double	I	m	The floating-point value to be classified.
<return>	int	O	m	FP_NAN when the parameter value is 'Not a Number'. FP_INFINITE when the parameter value is either plus or minus infinity. FP_ZERO when the parameter value is positive or negative zero. FP_SUBNORMAL when the parameter value is denormalized. Denormalized numbers have values that are too small (i.e., values very close to zero) to be represented otherwise. FP_NORMAL is returned for all other parameter values.

Example

```
int classification;
double value;
classification = fpclassify(value);
```

6.11.21 Builtin is_NaN

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Checks whether double value is not a number

The builtin is_NaN checks whether a double floating-point value is not a number. Not a number is specified by the IEEE 754 standard as an invalid double value (0x7FF8000000000000). This builtin is used to check if a double floating-point value has been initialized or if it has been set to a proper floating-point value.

Lexical structure

is_NaN(dvalue)

The lexical elements of the builtin is_NaN are shown in Table 301.

Table 301 – Builtin is_NaN

Parameter name	Type	Direction	Usage	Description
dvalue	double	I	m	The name of the variable.
<return>	long	O	m	One if the dvalue is equal to NaN, zero otherwise.

Example

```
long status;
double d;
status = display_message ((is_NaN(d) ? "|en|Value of \"d\": BAD":
"|en|Value of \"d\": %{d}"), 0, 0, 0);
```

6.11.22 Builtin log

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate natural logarithm

The builtin log returns the natural logarithm of a floating-point value.

Lexical structure

log(x)

The lexical elements of the builtin log are shown in Table 302.

Table 302 – Builtin log

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The natural logarithm.

6.11.23 Builtin log10

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate base 10 logarithm

The builtin exp returns the base 10 logarithm of a floating-point value.

Lexical structure

`log10(x)`

The lexical elements of the builtin `log10` are shown in Table 303.

Table 303 – Builtin `log10`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The base 10 logarithm.

6.11.24 Builtin `log2`

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate base 2 logarithm

The builtin `log2` returns the base 2 logarithm of a floating-point value.

Lexical structure

`log2(x)`

The lexical elements of the builtin `log2` are shown in Table 304.

Table 304 – Builtin `log2`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The base 2 logarithm.

6.11.25 Builtin `LongToByte`

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate long from bytes

The builtin `LongToByte` will provide a vector of unsigned char values representing the particular elements of a long value.

Lexical structure

`LongToByte(in, out1, out2, out3, out4)`

The lexical element of the builtin `LongToByte` is shown in Table 305.

Table 305 – Builtin LongToByte

Parameter name	Type	Direction	Usage	Description
in	long	I	m	The long value to convert.
out1	unsigned char	O	m	First element of the long value (most significant byte).
out2	unsigned char	O	m	Second element of the long value.
out3	unsigned char	O	m	Third element of the long value.
out4	unsigned char	O	m	Fourth element of the long value (least significant byte).
<return>	VOID	—	—	—

6.11.26 Builtin nan

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Return a "not a number" value

The builtin nan returns a "not a number" value as a double with a fractional bit pattern based on a string in the value parameter. If the character sequence begins with a quiet or signaling NaN prefix letter, then any following hex digits in the character sequence shall be valid for that type of NaN.

The string can be:

- "" (an empty string) generates the default NaN 0x7FF4000000000000.
- "Q" or "q" equals the default quiet NaN.
- "S" or "s" equals the default signaling NaN.
- "I_XXXXX" equals where I is either a "Q", "q", "S", or "s" followed by an underscore and the most significant hex digits of the fraction. Trailing zeros may be omitted.

Parsing of value string stops at the first improper character detected and a NaN whose fraction bits have been described by the string argument up to the point of the improper character is returned.

Lexical structure

nan(value)

The lexical elements of the builtin nan are shown in Table 306.

Table 306 – Builtin nan

Parameter name	Type	Direction	Usage	Description
value	DD_STRING	I	m	The "not a number" value to be returned. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	double	O	m	The specified value. Should an error occur in parsing nan_string then the NaN generated up to that point is returned.

Example

```
double nanval;
nanval = nan("");
```

6.11.27 Builtin NaN_value

Communication profiles: CS, FF, GPE, ISA100, PB, PN

Purpose: Return a "not a number" value

The builtin NaN_value returns the value of "not a number".

Lexical structure

```
NaN_value(value)
```

The lexical elements of the builtin NaN_value are shown in Table 307.

Table 307 – Builtin NaN_value

Parameter name	Type	Direction	Usage	Description
value	double	O	m	A pointer through which the value is returned.
<return>	long	O	m	The specified value.

Example

```
long status;
double nanval;
status = NaN_value(&nanval);
```

6.11.28 Builtin nanf

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Return a "not a number" value

The builtin nanf returns a "not a number" value as a float with a fractional bit pattern based on a string. If the character sequence begins with a quiet or signaling NaN prefix letter, then any following hex digits in the character sequence shall be valid for that type of NaN.

The string can be:

- "" (an empty string), generates the default NaN 0x7FA00000.
- "Q" or "q" equals the default quiet NaN.
- "S" or "s" equals the default signaling NaN.
- "I_XXXXXX" equals where I is either a "Q", "q", "S", or "s" followed by an underscore and the most significant hex digits of the fraction. Trailing zeros maybe omitted.

Parsing of value string stops at the first improper character detected and a NaN whose fraction bits have been described by the string argument up to the point of the improper character is returned.

Lexical structure

```
nanf(value)
```

The lexical elements of the builtin nanf are shown in Table 308.

Table 308 – Builtin nanf

Parameter name	Type	Direction	Usage	Description
value	DD_STRING	I	m	The "not a number" value to be returned. String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	float	O	m	The specified value. Should an error occur in parsing nan_string then the NaN generated up to that point is returned.

Example

```
float nanval;
nanval = nanf("");
```

6.11.29 Builtin pow

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate power

The builtin pow returns the power function of two floating-point values.

Lexical structure

```
pow(x, y)
```

The lexical elements of the builtin pow are shown in Table 309.

Table 309 – Builtin pow

Parameter name	Type	Direction	Usage	Description
x	double	I	m	The base.
y	double	I	m	The exponent.
<return>	double	O	m	The value of x raised to the y power.

6.11.30 Builtin round

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate nearest Integer

The builtin round rounds a floating-point value to the nearest integer.

Lexical structure

```
round(x)
```

The lexical elements of the builtin round are shown in Table 310.

Table 310 – Builtin round

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The rounded value.

Example

```
double v, value;
v = round(value);
```

6.11.31 Builtin ShortToByte

Communication profiles: CS, GPE, PB, PN

Purpose: Calculate short from byte

The builtin ShortToByte will provide a vector of unsigned char values representing the particular elements of a short value.

Lexical structure

```
ShortToByte(in, out1, out2)
```

The lexical elements of the builtin ShortToByte are shown in Table 311.

Table 311 – Builtin ShortToByte

Parameter name	Type	Direction	Usage	Description
in	short	I	m	The short value to convert.
out1	unsigned char	O	m	First element of the short value (most significant byte).
out2	unsigned char	O	m	Second element of the short value (least significant byte).
<return>	VOID	—	—	—

6.11.32 Builtin sin

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate sine

The builtin sin returns the sine a floating-point value (measured in radians).

Lexical structure

```
sin(x)
```

The lexical elements of the builtin sin are shown in Table 312.

Table 312 – Builtin sin

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The sine.

Example

```
double v, value;
v = sin(value);
```

6.11.33 Builtin sinh**Communication profiles:** CS, FF, GPE, HART, ISA100, PB, PN**Purpose:** Calculate hyperbolic sine

The builtin sinh returns the hyperbolic sine of a floating-point value.

Lexical structure`sinh(x)`

The lexical elements of the builtin sinh are shown in Table 313.

Table 313 – Builtin sinh

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The hyperbolic sine.

Example

```
double v, value;
v = sinh(value);
```

6.11.34 Builtin sqrt**Communication profiles:** CS, FF, GPE, HART, ISA100, PB, PN**Purpose:** Calculate square root

The builtin sqrt returns the square root of a floating-point value.

Lexical structure`sqrt(x)`

The lexical elements of the builtin sqrt are shown in Table 314.

Table 314 – Builtin sqrt

Parameter name	Type	Direction	Usage	Description
x	double	I	m	A floating-point value.
<return>	double	O	m	The square root.

Example

```
double v, value;
v = sqrt(value);
```

6.11.35 Builtin tan**Communication profiles:** CS, FF, GPE, HART, ISA100, PB, PN**Purpose:** Calculate tangent

The builtin tan returns the tangent of a floating-point value (measured in radians).

Lexical structure

`tan(x)`

The lexical elements of the builtin `tan` are shown in Table 315.

Table 315 – Builtin `tan`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The tangent.

Example

```
double v, value;
v = tan(value);
```

6.11.36 Builtin `tanh`

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Calculate hyperbolic tangent

The builtin `tanh` returns the hyperbolic tangent of a floating-point value.

Lexical structure

`tanh(x)`

The lexical elements of the builtin `tanh` are shown in Table 316.

Table 316 – Builtin `tanh`

Parameter name	Type	Direction	Usage	Description
<code>x</code>	double	I	m	A floating-point value.
<code><return></code>	double	O	m	The hyperbolic tangent.

Example

```
double v, value;
v = tanh(value);
```

6.12 List builtins**6.12.1 Builtin `get_date_lelem`**

Communication profiles: FF, ISA100

Purpose: Extracts a date value from a list

The builtin `get_date_lelem` extracts a date value from a LIST.

The `list_id` parameter specifies the item id of the list to be accessed, while the `index` parameter specifies the index of the list to be accessed.

If the `element_id` parameter references a simple variable, the `subelement_id` parameter is ignored. If the `element_id` parameter references a record, the `subelement_id` parameter specifies a member of the record. If the `element_id` parameter references an array parameter, the `subelement_id` parameter specifies an element of the array.

The value is returned via the data and size output parameters.

Lexical structure

```
get_date_lelem(list_id, index, element_id, subelement_id, data, size)
```

The lexical elements of the builtin `get_date_lelem` are shown in Table 317.

Table 317 – Builtin `get_date_lelem`

Parameter name	Type	Direction	Usage	Description
<code>list_id</code>	unsigned long	I	m	The item id of the list.
<code>index</code>	int	I	m	The index of the list containing the item to be extracted.
<code>element_id</code>	unsigned long	I	m	The item id of the element to be extracted.
<code>subelement_id</code>	unsigned long	I	m	The member id of the element to be extracted.
<code>data</code>	char *	O	m	The buffer in which the data is to be returned.
<code>size</code>	int *	O	m	The size of the data returned.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
int size;
char buf[8];
```

```
status = get_date_lelem(ITEM_ID(list), 0, ITEM_ID(var), 0, buf, &size);
```

6.12.2 Builtin `get_date_lelem2`

Communication profiles: FF, ISA100

Purpose: Extracts a date value from an embedded list

The builtin `get_date_lelem2` extracts a date value from a list that is embedded within another list.

The `list_id` parameter specifies the item id of the list to be accessed, while the `index` parameter specifies the index of the list to be accessed. The `embedded_list_id` parameter specifies the item id of the embedded list to be accessed, while the `embedded_list_index` parameter specifies the index of the embedded list to be accessed.

If the `element_id` parameter references a simple variable, the `subelement_id` parameter is ignored. If the `element_id` parameter references a record, the `subelement_id` parameter specifies a member of the record. If the `element_id` parameter references an array parameter, the `subelement_id` parameter specifies an element of the array.

The value is returned via the data and size output parameters.

Lexical structure

```
get_date_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id, data, size)
```

The lexical elements of the builtin `get_date_lelem2` are shown in Table 318.

Table 318 – Builtin `get_date_lelem2`

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
embedded_list_id	unsigned long	I	m	The item id of the embedded list.
embedded_list_index	int	I	m	The index of the embedded list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
data	char *	O	m	The buffer in which the data is to be returned.
size	int *	O	m	The size of the data returned.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
int size;
char buf[8];
```

```
status = get_date_lelem2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list), 12,
ITEM_ID(var), 0, buf, &size);
```

6.12.3 Builtin `get_double_lelem`

Communication profiles: FF, ISA100

Purpose: Extracts a double value from a list

The builtin `get_double_lelem` extracts a double value from a LIST.

The `list_id` parameter specifies the item id of the list to be accessed, while the `index` parameter specifies the index of the list to be accessed.

If the `element_id` parameter references a simple variable, the `subelement_id` parameter is ignored. If the `element_id` parameter references a record, the `subelement_id` parameter specifies a member of the record. If the `element_id` parameter references an array parameter, the `subelement_id` parameter specifies an element of the array.

Lexical structure

```
get_double_lelem(list_id, index, element_id, subelement_id)
```

The lexical element of the builtin `get_double_lelem` is shown in Table 319.

Table 319 – Builtin get_double_lelem

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	double	O	m	The specified value.

Example

```
double value;
value = get_double_lelem(ITEM_ID(list), 0, ITEM_ID(var), 0);
```

6.12.4 Builtin get_double_lelem2**Communication profiles: FF, ISA100****Purpose: Extracts a double value from an embedded list**

The builtin get_double_lelem2 extracts a double value from a list that is embedded within an another list.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed. The embedded_list_id parameter specifies the item id of the embedded list to be accessed, while the embedded_list_index parameter specifies the index of the embedded list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

```
get_double_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id)
```

The lexical elements of the builtin get_double_lelem2 are shown in Table 320.

Table 320 – Builtin get_double_lelem2

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
embedded_list_id	unsigned long	I	m	The item id of the embedded list.
embedded_list_index	int	I	m	The index of the embedded list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	double	O	m	The specified value.

Example

```
double value;
status = get_double_lelem2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list),
12, ITEM_ID(var), 0);
```

6.12.5 Builtin get_float_lelem**Communication profiles: FF, ISA100****Purpose: Extracts a float value from a list**

The builtin get_float_lelem extracts a float value from a LIST.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

```
get_float_lelem(list_id, index, element_id, subelement_id)
```

The lexical elements of the builtin get_float_lelem are shown in Table 321.

Table 321 – Builtin get_float_lelem

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	float	O	m	The specified value.

Example

```
float value;
value = get_float_lelem2(ITEM_ID(list), 0, ITEM_ID(var), 0);
```

6.12.6 Builtin get_float_lelem2**Communication profiles: FF, ISA100****Purpose: Extracts a float value from an embedded list**

The builtin `get_float_lelem2` extracts a float value from a list that is embedded within another list.

The `list_id` parameter specifies the item id of the list to be accessed, while the `index` parameter specifies the index of the list to be accessed. The `embedded_list_id` parameter specifies the item id of the embedded list to be accessed, while the `embedded_list_index` parameter specifies the index of the embedded list to be accessed.

If the `element_id` parameter references a simple variable, the `subelement_id` parameter is ignored. If the `element_id` parameter references a record, the `subelement_id` parameter specifies a member of the record. If the `element_id` parameter references an array parameter, the `subelement_id` parameter specifies an element of the array.

Lexical structure

```
get_float_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id)
```

The lexical elements of the builtin `get_float_lelem2` are shown in Table 322.

Table 322 – Builtin get_float_lelem2

Parameter name	Type	Direction	Usage	Description
<code>list_id</code>	unsigned long	I	m	The item id of the list.
<code>index</code>	int	I	m	The index of the list containing the item to be extracted.
<code>embedded_list_id</code>	unsigned long	I	m	The item id of the embedded list.
<code>embedded_list_index</code>	int	I	m	The index of the embedded list containing the item to be extracted.
<code>element_id</code>	unsigned long	I	m	The item id of the element to be extracted.
<code>subelement_id</code>	unsigned long	I	m	The member id of the element to be extracted.
<return>	float	O	m	The specified value.

Example

```
float value;
status = get_float_lelem2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list), 12,
ITEM_ID(var), 0);
```


6.12.7 Builtin get_signed_lelem

Communication profiles: FF, ISA100

Purpose: Extracts a signed integer value from a list

The builtin get_signed_lelem extracts a signed value from a LIST.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

get_signed_lelem(list_id, index, element_id, subelement_id)

The lexical elements of the builtin get_signed_lelem are shown in Table 323.

Table 323 – Builtin get_signed_lelem

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	l	m	The item id of the list.
index	int	l	m	The index of the list containing the item to be extracted.
element_id	unsigned long	l	m	The item id of the element to be extracted.
subelement_id	unsigned long	l	m	The member id of the element to be extracted.
<return>	long	o	m	The specified value.

Example

```
long value;
value = get_signed_lelem(ITEM_ID(list), 0, ITEM_ID(var), 0);
```

6.12.8 Builtin get_signed_lelem2

Communication profiles: FF, ISA100

Purpose: Extracts a signed integer value from an embedded list

The builtin get_signed_lelem2 extracts a signed value from a list that is embedded within another list.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed. The embedded_list_id parameter specifies the item id of the embedded list to be accessed, while the embedded_list_index parameter specifies the index of the embedded list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

```
get_signed_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id)
```

The lexical elements of the builtin `get_signed_lelem2` are shown in Table 324.

Table 324 – Builtin `get_signed_lelem2`

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
embedded_list_id	unsigned long	I	m	The item id of the embedded list.
embedded_list_index	int	I	m	The index of the embedded list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	long	O	m	The specified value.

Example

```
long value;
value = get_signed_lelem2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list), 12,
ITEM_ID(var), 0);
```

6.12.9 Builtin `get_string_lelem`

Communication profiles: FF, ISA100

Purpose: Extracts a string value from a list

The builtin `get_string_lelem` extracts a string from a LIST.

The `list_id` parameter specifies the item id of the list to be accessed, while the `index` parameter specifies the index of the list to be accessed.

If the `element_id` parameter references a simple variable, the `subelement_id` parameter is ignored. If the `element_id` parameter references a record, the `subelement_id` parameter specifies a member of the record. If the `element_id` parameter references an array parameter, the `subelement_id` parameter specifies an element of the array.

The value is returned via the string and length output parameters.

Lexical structure

```
get_string_lelem(list_id, index, element_id, subelement_id, string, length)
```

The lexical elements of the builtin `get_string_lelem` are shown in Table 325.

Table 325 – Builtin get_string_lelem

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
string	char *	O	m	The buffer in which the data is to be returned.
length	int *	O	m	The size of the data returned.
<return>	long	O	m	The specified value.

Example

```

long status;
int size;
char buf[256];
size = 256;
status = get_string_lelem(ITEM_ID(list), 0, ITEM_ID(var), 0, buf, &size);

```

6.12.10 Builtin get_string_lelem2**Communication profiles: FF, ISA100****Purpose: Extracts a string value from an embedded list**

The builtin get_string_lelem2 extracts a string from list that is embedded within another list.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed. The embedded_list_id parameter specifies the item id of the embedded list to be accessed, while the embedded_list_index parameter specifies the index of the embedded list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

The value is returned via the string and length output parameters.

Lexical structure

```

get_string_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id, string, length)

```

The lexical elements of the builtin get_string_lelem2 are shown in Table 326.

Table 326 – Builtin get_string_lelem2

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
embedded_list_id	unsigned long	I	m	The item id of the embedded list.
embedded_list_index	int	I	m	The index of the embedded list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
string	char*	O	m	The buffer in which the data is to be returned.
length	int*	O	m	The size of the data returned.
<return>	long	O	m	The specified value.

Example

```

long status;
int size;
char buf[256];
size = 256;
status = get_string_lelem2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list), 12
ITEM_ID(var), 0, buf, &size);

```

6.12.11 Builtin get_unsigned_lelem**Communication profiles: FF, ISA100****Purpose: Extracts a unsigned integer value from a list**

The builtin get_unsigned_lelem extracts an unsigned value from a LIST.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

```
get_unsigned_lelem(list_id, index, element_id, subelement_id)
```

The lexical elements of the builtin get_unsigned_lelem are shown in Table 327.

Table 327 – Builtin get_unsigned_lelem

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	unsigned long	O	m	The specified value.

Example

```
unsigned long value;
value = get_unsigned_lelem(ITEM_ID(list), 0, ITEM_ID(var), 0);
```

6.12.12 Builtin get_unsigned_lelem2**Communication profiles: FF, ISA100****Purpose: Extracts a unsigned integer value from an embedded list**

The builtin get_unsigned_lelem2 extracts an unsigned value from a list that is embedded within another list.

The list_id parameter specifies the item id of the list to be accessed, while the index parameter specifies the index of the list to be accessed. The embedded_list_id parameter specifies the item id of the embedded list to be accessed, while the embedded_list_index parameter specifies the index of the embedded list to be accessed.

If the element_id parameter references a simple variable, the subelement_id parameter is ignored. If the element_id parameter references a record, the subelement_id parameter specifies a member of the record. If the element_id parameter references an array parameter, the subelement_id parameter specifies an element of the array.

Lexical structure

```
get_unsigned_lelem2(list_id, index, embedded_list_id, embedded_list_index,
element_id, subelement_id)
```

The lexical elements of the builtin get_unsigned_lelem2 are shown in Table 328.

Table 328 – Builtin get_unsigned_lelem2

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The item id of the list.
index	int	I	m	The index of the list containing the item to be extracted.
embedded_list_id	unsigned long	I	m	The item id of the embedded list.
embedded_list_index	int	I	m	The index of the embedded list containing the item to be extracted.
element_id	unsigned long	I	m	The item id of the element to be extracted.
subelement_id	unsigned long	I	m	The member id of the element to be extracted.
<return>	unsigned long	O	m	Returns the specified value.

Example

```
unsigned long value;
status = get_unsigned_lelem2 (ITEM_ID(top_list), 0, ITEM_ID(embedded_list),
12, ITEM_ID(var), 0);
```

6.12.13 Builtin ListDeleteElementAt (version A)**Communication profiles: CS, GPE, HART, PB, PN****Purpose: Delete an element from a list.**

The builtin ListDeleteElementAt will delete an element from a list. If the list is a member of a FILE, the specified element will be removed from the persistent store.

Execution of ListDeleteElementAt causes the VARIABLE referenced by the COUNT attribute to be updated to match to the current number of elements in the LIST.

If the COUNT attribute resolves to an expression, then none of the VARIABLES referenced in the expression will be updated as a result of ListDeleteElementAt.

If the index parameter is less than zero or greater than the number of elements in the list, an error will be returned. If the list parameter does not refer to a LIST instance, an error will be returned.

Lexical structure

```
ListDeleteElementAt(list, index)
```

The lexical elements of the builtin ListDeleteElementAt are shown in Table 329.

Table 329 – Builtin ListDeleteElementAt

Parameter name	Type	Direction	Usage	Description
list	reference	I	m	The name of a LIST instance.
index	int	I	m	The index of the element to be deleted.
<return>	long	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the item was successfully removed from the list, and BI_ERROR if an error occurred.

6.12.14 Builtin ListDeleteElementAt (version B)**Communication profiles: FF, ISA100****Purpose: Delete an element from a list**

The builtin ListDeleteElementAt will delete an element from a list. If the list is a member of a FILE, the specified element will be removed from the persistent store.

Execution of ListDeleteElementAt causes the variable referenced by the COUNT attribute to be updated to match to the current number of elements in the LIST.

If the COUNT attribute resolves to an expression, then none of the variables referenced in the expression will be updated as a result of ListDeleteElementAt.

If the index parameter is less than zero or greater than the number of elements in the list, an error will be returned. If the list parameter does not refer to a LIST instance, an error will be returned.

Lexical structure

```
ListDeleteElementAt(list, index)
```

The lexical elements of the builtin ListDeleteElementAt are shown in Table 330.

Table 330 – Builtin ListDeleteElementAt

Parameter name	Type	Direction	Usage	Description
list	unsigned long	I	m	The item id of a LIST instance.
index	int	I	m	The index of the element to be deleted.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = ListDeleteElementAt(ITEM_ID(floatlist), 12);
```

6.12.15 Builtin ListDeleteElementAt2**Communication profiles: FF, ISA100****Purpose: Delete an element from an embedded list**

The builtin ListDeleteElementAt2 deletes elements from an embedded list. If the list is a member of a FILE, the specified element will be removed from the persistent store.

If the `list_id` parameter does not refer to a list, an error will be returned. If the `embedded_list_id` does not refer to a list embedded in the top-level list, an error will be returned. If the `index` parameter does not denote a valid element in the top-level list, an error will be returned. If the `embedded_index` and `embedded_count` parameters do not denote a valid range of elements of the embedded list, an error will be returned.

Only elements of the specified embedded list are removed. The top-level list is otherwise unaffected.

Lexical structure

```
ListDeleteElementAt2(list_id, list_index, embedded_list_id, embedded_index,
embedded_count)
```

The lexical elements of the builtin `ListDeleteElementAt2` are shown in Table 331.

Table 331 – Builtin ListDeleteElementAt2

Parameter name	Type	Direction	Usage	Description
<code>list_id</code>	unsigned long	I	m	The item of the list.
<code>list_index</code>	int	I	m	The index of the list containing the embedded list.
<code>embedded_list_id</code>	unsigned long	I	m	The item id of the embedded_list.
<code>embedded_index</code>	int	I	m	The index of the first element to delete.
<code>embedded_count</code>	int	I	m	The number of elements to delete.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = ListDeleteElementAt2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list),
12, 3);
```

6.12.16 Builtin ListInsert (version A)

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Insert element into a list

The builtin `ListInsert` will insert an element to a list. If the list is a member of a `FILE`, the specified element will be added to the persistent store.

Execution of `ListInsert` causes the `VARIABLE` referenced by the `COUNT` attribute to be updated to match to the current number of elements in the `LIST`.

If the `COUNT` attribute resolves to an expression, then none of the `VARIABLEs` referenced in the expression will be updated as a result of `ListInsert`.

If the `index` parameter is less than zero or greater than the capacity of the list, an error will be returned. If the `list` parameter does not refer to a `LIST` instance, an error will be returned.

Lexical structure

```
ListInsert(list, index, item)
```

The lexical elements of the builtin `ListInsert` are shown in Table 332.

Table 332 – Builtin ListInsert

Parameter name	Type	Direction	Usage	Description
list	reference	I	m	The name of a LIST instance.
index	int	I	m	The index of the requested element.
item	reference	I	m	The name of the item to be inserted.
<return>	long	O	m	One of the return codes specified in Table 377.

6.12.17 Builtin ListInsert (version B)**Communication profiles: FF, ISA100****Purpose: Insert element into a list**

The builtin ListInsert will insert an element to a list. If the list is a member of a FILE, the specified element will be added to the persistent store.

Execution of ListInsert causes the variable referenced by the COUNT attribute to be updated to match to the current number of elements in the LIST.

If the COUNT attribute resolves to an expression, then none of the variables referenced in the expression will be updated as a result of ListInsert.

If the index parameter is less than zero or greater than the capacity of the list, an error will be returned. If the list parameter does not refer to a LIST instance, an error will be returned.

Lexical structure

```
ListInsert(list, index, item)
```

The lexical elements of the builtin ListInsert are shown in Table 333.

Table 333 – Builtin ListInsert

Parameter name	Type	Direction	Usage	Description
list	unsigned long	I	m	The item id of a LIST instance.
index	int	I	m	The index of the requested element.
item	unsigned long	I	m	The item id of the item to be inserted.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = ListInsert(ITEM_ID(floatlist), 12, ITEM_ID(floatvar));
```

6.12.18 Builtin ListInsert2**Communication profiles: FF, ISA100****Purpose: Insert element into an embedded list**

The builtin ListInsert2 inserts an element into a list that is embedded within another list. If the list is a member of a FILE, the specified element will be added to the persistent store.

If the `list_id` parameter does not refer to a list, an error will be returned. If the `embedded_list_id` does not refer to a list embedded in the top-level list, an error will be returned. If the `list_index` parameter does not denote a valid element in the top-level list, an error will be returned. If the `embedded_index` and `embedded_count` parameters do not denote a valid range of elements of the embedded list, an error will be returned.

Only elements of the specified embedded list are removed. The top-level list is otherwise unaffected.

Lexical structure

`ListInsert2(list_id, list_index, embedded_list_id, embedded_index, item_id)`

The lexical elements of the builtin `ListInsert2` are shown in Table 334.

Table 334 – Builtin ListInsert2

Parameter name	Type	Direction	Usage	Description
<code>list_id</code>	unsigned long	I	m	The item of the list.
<code>list_index</code>	int	I	m	The index of the list containing the embedded list.
<code>embedded_list_id</code>	unsigned long	I	m	The item id of the embedded list.
<code>embedded_index</code>	int	I	m	The index where the item should be inserted.
<code>item_id</code>	unsigned long	I	m	The item id of the item to be inserted.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = ListDeleteElementAt2(ITEM_ID(top_list), 0, ITEM_ID(embedded_list),
12, 3);
```

6.13 Translation builtins

6.13.1 Builtin DICT_ID

Communication profiles: FF, ISA100

Purpose: Convert a standard text dictionary name into a standard text dictionary id

The builtin `DICT_ID` can be used to specify a text dictionary id for calls to `get_stddict_string`. It converts a text dictionary name into a text dictionary id.

A text dictionary entry consists of three parts: a unique id, a name, and a multilingual description. This function retrieves the unique id using the text dictionary name.

Lexical structure

`DICT_ID(name)`

The lexical elements of the builtin `DICT_ID` are shown in Table 335.

Table 335 – Builtin DICT_ID

Parameter name	Type	Direction	Usage	Description
name	reference	I	m	The name of the dictionary string.
<return>	long	O	m	The dictionary id for the named string.

Example:

```

long status;
char msg[256];
int len;
status = get_stddict_string(DICT_ID(square_root), msg, len);

```

6.13.2 Builtin get_block_instance_by_object_index

Communication profiles: FF, ISA100

Purpose: Return occurrence number of the BLOCK_A instance at the specified object index

The builtin `get_block_instance_by_object_index` returns the occurrence number of the BLOCK_A instance at the specified object index. The object index parameter specifies the object index of one of the device's block objects.

Lexical structure

```
get_block_instance_by_object_index(obj_idx, blk_num)
```

The lexical elements of the builtin `get_block_instance_by_object_index` are shown in Table 336.

Table 336 – Builtin get_block_instance_by_object_index

Parameter name	Type	Direction	Usage	Description
obj_idx	unsigned long	I	m	The object index of the BLOCK_A instance to be returned.
blk_num	unsigned long	O	m	The occurrence number of the specified BLOCK_A instance.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```

long status;
unsigned long index;
unsigned long blockIds[1];
unsigned long blockInstances[1];
unsigned long paramIds[1];
unsigned long memberIds[1];

status = get_block_instance_by_object_index(1500, &index);

blockIds[0] = ITEM_ID(transducer_block);
blockInstances[0] = index;
paramIds[0] = ITEM_ID(PARAM.MODE_BLK);
memberIds[0] = MEMBER_ID(ACTUAL);

status = display_message2("TRANSDUCER MODE = %{0}", blockIds,
blockInstances, paramIds, memberIds, 1);

```

6.13.3 Builtin get_block_instance_by_tag

Communication profiles: FF, ISA100

Purpose: Return occurrence number of the BLOCK_A instance with the specified tag

The builtin `get_block_instance_by_tag` returns the occurrence number of the `BLOCK_A` instance with the specified tag. The `blockId` and `blockTag` parameters specify the DD item id and block tag elements of one of the device's block objects, respectively. If the `blockId` parameter is zero, the occurrence number of the block whose block tag matches the `blockTag` parameter will be returned. Otherwise, the occurrence number of the block whose DD item id matches the `blockId` parameter and whose block tag matches the `blockTag` parameter will be returned. If multiple blocks meet the specified criteria, the occurrence number of the block with the smallest object index will be returned.

Lexical structure

`get_block_instance_by_tag(blockId, blockTag, blk_num)`

The lexical elements of the builtin `get_block_instance_by_tag` are shown in Table 337.

Table 337 – Builtin get_block_instance_by_tag

Parameter name	Type	Direction	Usage	Description
<code>blockId</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> instance to be returned.
<code>blockTag</code>	char[]	I	m	The tag of the <code>BLOCK_A</code> instance to be returned.
<code>blk_num</code>	unsigned long	O	m	The occurrence number of the specified <code>BLOCK_A</code> instance.
<return>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long index;
unsigned long blockIds[1];
unsigned long blockInstances[1];
unsigned long paramIds[1];
unsigned long memberIds[1];

status = get_block_instance_tag(ITEM_ID(__analog_input_block), "AI1",
&index);

blockIds[0] = ITEM_ID(__analog_input_block);
blockInstances[0] = index;
paramIds[0] = ITEM_ID(PARAM.MODE_BLK);
memberIds[0] = MEMBER_ID(ACTUAL);

status = display_message2("AI1 MODE = %{0}", blockIds, blockInstances,
paramIds, memberIds, 1);
```

6.13.4 Builtin `get_block_instance_count`

Communication profiles: FF, ISA100

Purpose: Return number of blocks instantiated in the device

The builtin `get_block_instance_count` returns the number of `BLOCK_A` instances that are instantiated in the device. The `blockId` parameter specifies the DD item id element of one or more of the device's block objects. If the `blockId` parameter is zero, the total number of blocks currently instantiated in the device will be returned. Otherwise, the number of blocks whose DD item id matches the `blockId` parameter will be returned.

Lexical structure

`get_block_instance_count(blockId, count)`

The lexical elements of the builtin `get_block_instance_count` are shown in Table 338.

Table 338 – Builtin `get_block_instance_count`

Parameter name	Type	Direction	Usage	Description
<code>blockId</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> instances to be counted.
<code>count</code>	unsigned long	O	m	The number of <code>BLOCK_A</code> instances of the specified type that are instantiated in the device.
<code><return></code>	long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
unsigned long aiBlockCount;
int i;
status = get_block_instance_count(ITEM_ID(__analog_input_block),
&aiBlockCount);
for (i = 0; i < aiBlockCount; ++i)
{
    /* do something each to AI block */
}
```

6.13.5 Builtin `get_resolve_status`

Communication profiles: FF, ISA100

Purpose: Get most recent error codes

The builtin `get_resolve_status` returns the codes for errors generated by the most recent call of builtin `resolve_array_ref`, `resolve_block_ref`, `resolve_parm_ref`, `resolve_param_list_ref` or `resolve_record_ref`.

The reference resolution builtins, those starting with “`resolve_`”, resolve the possibly dynamic id of an element of a `BLOCK_A`, `PARAMETER`, `PARAMETER_LIST`, `VALUE_ARRAY`, `REFERENCE_ARRAY`, `COLLECTION`, or `RECORD`. If any of these resolve reference builtins encounters an error, it returns a zero for the id.

If the calling method wants to know what the error was, then it calls the builtin `get_resolve_status` to get the error number. This builtin is used for debugging purposes when creating methods.

Lexical structure

```
get_resolve_status(VOID)
```

The lexical element of the builtin `get_resolve_status` is shown in Table 339.

Table 339 – Builtin `get_resolve_status`

Parameter name	Type	Direction	Usage	Description
<return>	unsigned long	O	m	One of the return codes specified in Table 376.

Example

```
long status;
status = get_resolve_status();
```

6.13.6 Builtin `ITEM_ID`

Communication profiles: FF, ISA100

Purpose: Specify the item ID of a VARIABLE

The builtin `ITEM_ID` can be used in methods to specify the item id of a variable for calls to other.

Lexical structure

```
ITEM_ID(name)
```

The lexical elements of the builtin `ITEM_ID` are shown in Table 340.

Table 340 – Builtin `ITEM_ID`

Parameter name	Type	Direction	Usage	Description
name	reference	I	m	The name of the variable.
<return>	unsigned long	O	m	The VARIABLE's item id.

Example

```
long duration_size, status;
char duration_value[6];

status = put_date_value(ITEM_ID(PARAM.CALIBRATION), MEMBER_ID(DURATION),
duration_value, duration_size);
```

6.13.7 Builtin `MEMBER_ID`

Communication profiles: FF, ISA100

Purpose: Specify member id of a VARIABLE

The builtin `MEMBER_ID` can be used in methods to specify the member id of a VARIABLE for calls to other builtins. It returns the member id of a BLOCK PARAMETER, PARAMETER_LIST, RECORD, REFERENCE_ARRAY, or COLLECTION.

Lexical structure

```
MEMBER_ID(name)
```

The lexical elements of the builtin MEMBER_ID are shown in Table 341.

Table 341 – Builtin MEMBER_ID

Parameter name	Type	Direction	Usage	Description
name	reference	I	m	The name of an item member.
<return>	unsigned long	O	m	The referenced member id.

Example

```
long duration_size, status;
char duration_value[6];
```

```
status = put_date_value(ITEM_ID(PARAM.CALIBRATION), MEMBER_ID(DURATION),
duration_value, duration_size);
```

6.13.8 Builtin ObjectReference

Communication profiles: CS, GPE, PB, PN

Purpose: Get reference to a COMPONENT instance

The builtin ObjectReference returns a reference to a COMPONENT instance. The return value can be used to access data objects and methods of other COMPONENT instances.

NOTE For example, in an EDD of a remote I/O head station, it is possible to make a consistency check with the modules using ObjectReference(CHILD).

Lexical structure

ObjectReference(navigation)

The lexical elements of the builtin ObjectReference are shown in Table 342.

Table 342 – Builtin ObjectReference

Parameter name	Type	Direction	Usage	Description
navigation	One of the following values: SELF CHILD FIRST LAST PARENT NEXT PREV	I	m	See IEC 61804-3 for the direction of navigation.
<return>	OBJECT_REFERENCE	O	m	A reference to a COMPONENT instance.

6.13.9 Builtin resolve_array_ref

Purpose: Gets the item id of an array or list element

Communication profiles: HART

The builtin `resolve_array_ref` is used internally to specify a reference to an array or list element. Method developers shall not call `resolve_array_ref` directly, but rather any reference by a method developer to `name[x]` may be translated into `resolve_array_ref(name, x)` internally.

The builtin `resolve_array_ref` takes the name of the array or list and an index and resolves them into the item id of the indexed element referenced. This builtin can be used to resolve array references that are passed to builtins.

If an element of the array or list has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Communication profiles: FF, ISA100

The builtin `resolve_array_ref` is used by method developers when parsing things such as `item_array_name`.

The builtin `resolve_array_ref` takes the item id of the `REFERENCE_ARRAY` and an index and resolves them into the item id of the `REFERENCE_ARRAY` element referenced. This builtin can be used to fill arrays that are passed to display builtins.

The parameters `id` and `member_id` are specified using the builtins `ITEM_ID` and `MEMBER_ID`.

If a member of a `REFERENCE_ARRAY` has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Lexical structure

`resolve_array_ref(id, member_id)`

The lexical elements of the builtin `resolve_array_ref` are shown in Table 343.

Table 343 – Builtin resolve_array_ref

Parameter name	Type	Direction	Usage	Description
<code>id</code>	unsigned long	I	m	The item id of the array or list.
<code>member_id</code>	unsigned long	I	m	The index of the requested element.
<code><return></code>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
long status;
unsigned long ids[1];
ids[0] = resolve_array_ref(ITEM_ID(trend_item_array), LAST_SECOND);
status = display_message("|en|Last second: %[d]{0}\n", ids, 0, 1);
```


6.13.10 Builtin resolve_array_ref2

Communication profiles: FF, ISA100

Purpose: Gets the item id of a reference_array element

The builtin resolve_array_ref2 is used by method developers when parsing things such as BLOCK_A[blk_num].array[index]. The builtin resolve_array_ref2 takes the item id of the REFERENCE_ARRAY and an index and resolves them into the item id of the VALUE_ARRAY element referenced. This builtin can be used to fill arrays that are passed to display builtins.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID and the name or reference of the variable in question.

If a member of a VALUE_ARRAY has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Lexical structure

resolve_array_ref2(blk_id, blk_num, id, member_id)

The lexical elements of the builtin resolve_array_ref2 are shown in Table 344

Table 344 – Builtin resolve_array_ref2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A being resolved.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A being resolved.
id	unsigned long	I	m	The item id of the VALUE_ARRAY.
member_id	unsigned long	I	m	The index of the requested VALUE_ARRAY element.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin get_resolve_status may be used to access the actual error code.

Example

```
long status;
unsigned long ids[1];
ids[0] = resolve_array_ref2(ITEM_ID(block_name), 0,
ITEM_ID(trend_item_array), LAST_SECOND);
status = display_message("|en|Last second: %[d]{0}\n", ids, 0, 1);
```

6.13.11 Builtin resolve_block_ref

Communication profiles: FF

Purpose: Gets item id for BLOCK_A record member

The builtin resolve_block_ref is used by method developers when parsing things such as BLOCK_A.name. The builtin resolve_block_ref takes the member id of a member of a block's characteristic record and returns the item id for the resolved element.

The parameter `member_id` is specified using the `MEMBER_ID` builtin and the name or reference of the variable in question.

If a member of the characteristic record has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Lexical structure

```
resolve_block_ref(member_id)
```

The lexical elements of the builtin `resolve_block_ref` are shown in Table 345.

Table 345 – Builtin `resolve_block_ref`

Parameter name	Type	Direction	Usage	Description
<code>member_id</code>	unsigned long	I	m	The member id of <code>BLOCK_A</code> .
<code><return></code>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_block_ref(MEMBER_ID(BLK_CLASS));
```

6.13.12 Builtin `resolve_block_ref2`

Communication profiles: FF

Purpose: Gets item id for specific `BLOCK_A` record member

The builtin `resolve_block_ref2` is used by method developers when parsing things such as `BLOCK_A[blk_num].name`. The builtin `resolve_block_ref2` takes item id of a block, the occurrence number of the block within the device, and the member id of a member of a block's characteristic record and returns the item id for the resolved element.

The parameter `member_id` is specified using the `MEMBER_ID` builtin and the name or reference of the variable in question.

If a member of the characteristic record has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Lexical structure

```
resolve_block_ref2(blk_id, blk_num, member_id)
```

The lexical elements of the builtin `resolve_block_ref2` are shown in Table 346.

Table 346 – Builtin resolve_block_ref2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A being resolved.
blk_num	unsigned long	I	m	The occurrence number of the BLOCK_A being resolved.
member_id	unsigned long	I	m	The member id of BLOCK_A.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin get_resolve_status may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_block_ref2(ITEM_ID(block_name), 0, MEMBER_ID(BLK_CLASS));
```

6.13.13 Builtin resolve_list_ref**Communication profiles: FF, ISA100****Purpose: Return the type of a list**

The builtin resolve_list_ref returns the type of a LIST.

Lexical structure

```
resolve_list_ref(list_id)
```

The lexical elements of the builtin resolve_list_ref are shown in Table 347.

Table 347 – Builtin resolve_list_ref

Parameter name	Type	Direction	Usage	Description
list_id	unsigned long	I	m	The member id of the LIST.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin get_resolve_status may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_list_ref(ITEM_ID(list_name));
```

6.13.14 Builtin resolve_local_ref**Communication profiles: FF, ISA100****Purpose: Returns the item id of a local parameter**

The builtin resolve_local_ref is used by method developers when parsing things such as LOCAL_PARAM.name. It takes the member id of a LOCAL_PARAMETERS element and returns the item id of the completely resolved element.

The parameter `member_id` is specified using the `MEMBER_ID` builtin and the name or reference of the variable in question.

Lexical structure

```
resolve_local_ref(member_id)
```

The lexical elements of the builtin `resolve_local_ref` are shown in Table 348.

Table 348 – Builtin `resolve_local_ref`

Parameter name	Type	Direction	Usage	Description
<code>member_id</code>	unsigned long	I	m	The member id of the local parameter.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_local_ref(MEMBER_ID(LOCAL_PARAM.LOCAL1));
```

6.13.15 Builtin `resolve_local_ref2`

Communication profiles: FF, ISA100

Purpose: Returns the item id of a local parameter of a specific block

The builtin `resolve_local_ref2` is used by method developers when parsing things such as `BLOCK_A[blk_num].LOCAL_PARAM.name`. It takes the member id of a `BLOCK_A`, the occurrence number of a `BLOCK_A`, the member id of a `LOCAL_PARAMETERS` element and returns the item id of the completely resolved element.

The parameters `blk_id` and `member_id` are specified using the `MEMBER_ID` builtin.

Lexical structure

```
resolve_local_ref2(blk_id, blk_num, member_id)
```

The lexical elements of the builtin `resolve_local_ref2` are shown in Table 349.

Table 349 – Builtin `resolve_local_ref2`

Parameter name	Type	Direction	Usage	Description
<code>blk_id</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> being resolved.
<code>blk_num</code>	unsigned long	I	m	The occurrence number of the <code>BLOCK_A</code> being resolved.
<code>member_id</code>	unsigned long	I	m	The member id of the local parameter.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_local_ref2(ITEM_ID(block_name), 0,
MEMBER_ID(LOCAL_PARAM.LOCAL1));
```

6.13.16 Builtin resolve_param_list_ref**Communication profiles: FF, ISA100****Purpose: Resolve member id to item id**

The builtin `resolve_param_list_ref` resolves the `member_id` reference generated for a `PARAMETER_LIST` element into an item id, which is then used in builtin calls.

The builtin `resolve_param_list_ref` is used by method developers when parsing things such as a `PARAM_LIST` name. It takes the member id of a `PARAMETER_LIST` element and returns the item id of the completely resolved element.

The parameter `member_id` is specified using the `MEMBER_ID` builtin and the name or reference of the variable in question. If a member of the parameter list has a conditional dependency (the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

Lexical structure

```
resolve_param_list_ref(member_id)
```

The lexical elements of the builtin `resolve_param_list_ref` are shown in Table 350.

Table 350 – Builtin resolve_param_list_ref

Parameter name	Type	Direction	Usage	Description
<code>member_id</code>	unsigned long		m	The member id of the parameter list.
<code><return></code>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_param_list_ref(MEMBER_ID(PARAM_LIST.DIS1.VL_PV));
```

6.13.17 Builtin resolve_param_ref**Communication profiles: FF, ISA100****Purpose: Gets the item id of a parameter**

The builtin `resolve_param_ref` is used by method developers when parsing things such as `PARAM.name`. It takes the member id of a `PARAMETER` element and returns the item id of the completely resolved element.

The parameter `member_id` is specified using the `MEMBER_ID` builtin.

Lexical structure

```
resolve_param_ref(member_id)
```

The lexical elements of the builtin `resolve_param_ref` are shown in Table 351.

Table 351 – Builtin `resolve_param_ref`

Parameter name	Type	Direction	Usage	Description
<code>member_id</code>	unsigned long	I	m	The member id of the parameter.
<code><return></code>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_param_ref(MEMBER_ID(PARAM.OUT_STATE));
```

6.13.18 Builtin `resolve_param_ref2`

Communication profiles: FF, ISA100

Purpose: Gets the item id of a parameter of a specific block

The builtin `resolve_param_ref2` is used by method developers when parsing things such as `BLOCK_A[blk_num].PARAM.name`. It takes the member id of a `BLOCK_A`, an occurrence number, and a `PARAMETER` element and returns the item id of the completely resolved element.

The block name and parameter `member_id` are specified using the `MEMBER_ID`.

Lexical structure

```
resolve_param_ref2(blk_id, blk_num, member_id)
```

The lexical elements of the builtin `resolve_param_ref2` are shown in Table 352.

Table 352 – Builtin `resolve_param_ref2`

Parameter name	Type	Direction	Usage	Description
<code>blk_id</code>	unsigned long	I	m	The item id of the <code>BLOCK_A</code> being resolved.
<code>blk_num</code>	unsigned long	I	m	The occurrence number of the <code>BLOCK_A</code> being resolved.
<code>member_id</code>	unsigned long	I	m	The member id of the parameter.
<code><return></code>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin <code>get_resolve_status</code> may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_param_ref2(ITEM_ID(block_name), 0, MEMBER_ID(PARAM.LOCAL1));
```

6.13.19 Builtin resolve_record_ref

Communication profiles: FF, ISA100

Purpose: Returns the item id of a parameter

The builtin resolve_record_ref is used by method developers when parsing things such as record_name.member_name. It takes the item id of a record and the member id of a record element or member and returns the item id of the completely resolved element.

If a member of a record has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

The parameters id and member_id are specified using the builtins ITEM_ID and MEMBER_ID.

Lexical structure

```
resolve_record_ref(id, member_id)
```

The lexical elements of the builtin resolve_record_ref are shown in Table 353.

Table 353 – Builtin resolve_record_ref

Parameter name	Type	Direction	Usage	Description
id	unsigned long	I	m	The item id of the RECORD being resolved.
member_id	unsigned long	I	m	The member id of the record member.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin get_resolve_status may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_record_ref(ITEM_ID(PARAM.PV), MEMBER_ID(VALUE));
```

6.13.20 Builtin resolve_record_ref2

Communication profiles: FF, ISA100

Purpose: Returns the item id of a parameter of a specific block

The builtin resolve_record_ref2 is used by method developers when parsing things such as BLOCK_A[blk_num].record_name.member_name. It takes the item id of a block, the occurrence number of a block, the item id of a record and the member id of a record element or member and returns the item id of the completely resolved element.

If a member of a record has a conditional dependency (that is, the definition of an element is dependent on another value), the dependencies are taken into account during the resolution.

The parameters blk_id, id and member_id are specified using the builtins ITEM_ID and MEMBER_ID.

Lexical structure

```
resolve_record_ref2(blk_id, blk_num, id, member_id)
```

The lexical elements of the builtin resolve_record_ref2 are shown in Table 354.

Table 354 – Builtin resolve_record_ref2

Parameter name	Type	Direction	Usage	Description
blk_id	unsigned long	I	m	The item id of the BLOCK_A being resolved.
blk-num	unsigned long	I	m	The occurrence number of the BLOCK_A being resolved.
id	unsigned long	I	m	The item id of the RECORD being resolved.
member_id	unsigned long	I	m	The member id of the record member.
<return>	unsigned long	O	m	The item id of the resolved reference, or zero for an error. In the case of an error, builtin get_resolve_status may be used to access the actual error code.

Example

```
unsigned long id;
id = resolve_record_ref2(ITEM_ID(block_name), 0, ITEM_ID(record_name),
MEMBER_ID(REC_MEM_NAME));
```

6.13.21 Builtin VARID

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Return numeric identifier for VARIABLE

The builtin VARID will return the numeric identifier for the variable specified. A valid variable name shall be provided. Each variable in the device description is assigned a unique numeric identifier. This function is to be used when the identifier of a variable either needs to be stored in a temporary buffer, or needs to be sent as a parameter to another builtin.

Lexical structure

```
VARID(variable_name)
```

The lexical elements of the builtin VARID are shown in Table 355.

Table 355 – Builtin VARID

Parameter name	Type	Direction	Usage	Description
variable_name	reference	I	m	The identifier of the VARIABLE.
<return>	int	O	m	The variable identifier.

6.14 Block transfer builtins**6.14.1 Builtin abortTransferPort**

Communication profiles: HART

Purpose: Stop block transfer

The abortTransferPort stops the Block Data Transfer indicated by the handle.

Lexical structure

```
abortTransferPort(handle)
```


The lexical elements of the builtin abortTransferPort are shown in Table 356.

Table 356 – Builtin abortTransferPort

Parameter name	Type	Direction	Usage	Description
handle	unsigned int	I	m	The handle for the block transfer.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS when the transfer aborts or BI_ERROR if a transfer could not be aborted (e.g., the port was not actually open).

6.14.2 Builtin closeTransferPort

Communication profiles: HART

Purpose: Close port for block transfer

The closeTransferPort function closes the block transfer indicated by the handle.

Lexical structure

closeTransferPort(handle)

The lexical elements of the builtin closeTransferPort are shown in Table 357.

Table 357 – Builtin closeTransferPort

Parameter name	Type	Direction	Usage	Description
handle	unsigned int	I	m	The handle for the block transfer.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if successful, BI_PORT_IN_USE if a transfer is in process, or BI_ERROR if another problem is encountered (e.g., the port was not open).

6.14.3 Builtin fGetByte

Communication profiles: HART

Purpose: Access BLOB content

The builtin fGetByte allows access to BLOB contents for validation purposes only. It allows to access specified bytes in the BLOB to perform basic verification of the BLOB prior to its transmission, via block transfer, to the field device.

Lexical structure

fGetByte(blob_item, position, value_read)

The lexical elements of the builtin fGetByte are shown in Table 358.

Table 358 – Builtin fGetByte

Parameter name	Type	Direction	Usage	Description
blob_item	DD_ITEM &	I	m	The BLOB from which the byte shall be retrieved.
position	unsigned long long	I	m	The position relative to the start from which to read.
value_read	unsigned char &	O	m	Value from the BLOB.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the byte was successfully retrieved from the BLOB or BI_ERROR otherwise. If the return code is BI_SUCCESS then value_read shall contain a copy of the byte the blob_item located at the position parameter.

6.14.4 Builtin get_transfer_status**Communication profiles: HART****Purpose: Get status for ongoing block transfer**

The builtin get_transfer_status function is used to poll for completion of the DD_ITEM currently being transferred. It gets the latest information pertaining to the data transfer on the specified port and in the specified direction. The transfer_status array contains the response code, function code, master-byte-count, and device-byte-count from the latest Command 112 or 111 transaction.

Lexical structure

```
get_transfer_status(handle, direction, transfer_status)
```

The lexical elements of the builtin get_transfer_status are shown in Table 359.

Table 359 – Builtin get_transfer_status

Parameter name	Type	Direction	Usage	Description
handle	unsigned int	I	m	The handle for the block transfer
direction	unsigned char	I	m	Specifies the direction being polled: 0x01 for read 0x02 for write 0x03 for read and write
transfer_status	unsigned int[4]	O	m	The returned transfer status containing the following information: • response code; • function code; • master-bytecount; • device-byte-count.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the function was successful, and BI_ERROR if the function fails (e.g., if the port is not open). BI_CONTINUE will be returned when transfer has been started and is occurring. BI_RETRY is returned if the same DD_ITEM is already in the process of being transferred.

6.14.5 Builtin openTransferPort

Communication profiles: HART

Purpose: Open port for block transfer

The builtin openTransferPort opens the block transfer port using Command 111.

Lexical structure

`openTransferPort(port, handle)`

The lexical elements of the builtin openTransferPort are shown in Table 360.

Table 360 – Builtin openTransferPort

Parameter name	Type	Direction	Usage	Description
port	unsigned char	I	m	The block transfer port number.
handle	unsigned int	O	m	The resulting handle. Only valid if this builtin returns BI_SUCCESS.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the port successfully opened, and BI_ERROR otherwise.

6.14.6 Builtin readItemFromDevice

Communication profiles: HART

Purpose: Read DD_ITEM from device via block transfer

This builtin reads an item from an open block transfer indicated by the handle.

The DD_ITEM may be a compound item but shall contain only data.

As data is received from the field device it is buffered. When readItemFromDevice is called, the data is copied into the DD_Item. Upon completing the read, any excess data is retained in the buffer and is the first data consumed in the next read.

The PRE_READ_ACTIONS and POST_READ_ACTIONS attributes on the data items to be read from the device shall be ignored.

Lexical structure

`readItemFromDevice(handle, dd_item)`

The lexical elements of the builtin readItemFromDevice are shown in Table 361.

Table 361 – Builtin readItemFromDevice

Parameter name	Type	Direction	Usage	Description
handle	unsigned int	I	m	The handle of the block transfer.
dd_item	DD_ITEM &	O	m	The DD item to read.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the function was successful, and BI_ERROR if the function fails (e.g., if the port is not open). BI_CONTINUE will be returned when transfer has been started and is occurring. BI_RETRY is returned if the same DD-Item is already in the process of being transferred.

6.14.7 Builtin writeItemToDevice

Communication profiles: HART

Purpose: Write DD_ITEM to device via block transfer

The builtin writeItemToDevice writes an item from an open block transfer indicated by the handle. The DD_ITEM may be a compound item but shall contain only data. If the DD_ITEM is transferred in a single command 112 transaction, the builtin writeItemToDevice shall return BI_SUCCESS.

The PRE_WRITE_ACTIONS and POST_WRITE_ACTIONS attributes on the data items to be written to the device shall be ignored.

Lexical structure

```
writeItemToDevice(handle, dd_item)
```

The lexical elements of the builtin writeItemToDevice are shown in Table 362.

Table 362 – Builtin writeItemToDevice

Parameter name	Type	Direction	Usage	Description
handle	unsigned int	I	m	The handle of the block transfer.
dd_item	DD_ITEM &	I	m	The DD item to write.
<return>	int	O	m	One of the return codes specified in Table 377. BI_SUCCESS if the function was successful, and BI_ERROR if the function fails (e.g., if the port is not open). BI_CONTINUE will be returned when transfer has been started and is occurring. BI_RETRY is returned if the same DD-Item is already in the process of being transferred.

6.15 File builtins

6.15.1 Builtin browseIdentity

Communication profiles: HART

Purpose: Select BLOB

The BLOB declaration may not have a declared IDENTITY. The browseIdentity function allows the user to select the BLOB using a EDD application specific mechanism.

Lexical structure

```
browseIdentity(blob_item)
```

The lexical elements of the builtin browseIdentity are shown in Table 363.

Table 363 – Builtin browseIdentity

Parameter name	Type	Direction	Usage	Description
blob_item	DD_ITEM &	I	m	The BLOB for which the identity is browsed.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the BLOB was identified by the user or BI_ERROR otherwise. (e.g., the user canceled the browse, the file could not be accessed).

6.15.2 Builtin re_read_file

Communication profiles: HART

Purpose: Force reread of FILE

The builtin re_read_file causes the FILE's persistent storage area to be reread and update the values for all of the data items in the FILE members list.

Lexical structure

re_read_file(file_item)

The lexical elements of the builtin re_read_file are shown in Table 364.

Table 364 – Builtin re_read_file

Parameter name	Type	Direction	Usage	Description
file_item	DD_ITEM &	I	m	The FILE to read.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the execution is successful. BI_ERROR if the DD_ITEM was not of type FILE or another error was encountered.

6.15.3 Builtin re_write_file

Communication profiles: HART

Purpose: Force rewrite of FILE

The builtin re_write_file causes the values from the data items in the FILE members list to be rewritten to the FILE's persistent storage area.

A FILE that has a both a UPDATE ACTIONS and a METHOD using Building re_write_file() will have the actions executed before the builtin re_write_file() function returns.

Lexical structure

re_write_file(file_item)

The lexical elements of the builtin re_write_file are shown in Table 365.

Table 365 – Builtin re_write_file

Parameter name	Type	Direction	Usage	Description
file_item	DD_ITEM &	I	m	The FILE to write.
<return>	int	O	m	One of the return codes specified in Table 377. Returns BI_SUCCESS if the execution is successful. BI_ERROR if the DD_ITEM was not of type FILE or another error was encountered.

6.16 Identification builtins

6.16.1 Builtin GET_DD_REVISION

Communication profiles: CS, GPE, PB, PN

Purpose: Get DD revision from identification

The builtin GET_DD_REVISION gets the DD_REVISION from the currently used device description as stated in the identification of the device description with the keyword DD_REVISION.

Lexical structure

GET_DD_REVISION (VOID)

The lexical element of the builtin GET_DD_REVISION is shown in Table 366.

Table 366 – Builtin GET_DD_REVISION

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	The current DD_REVISION.

6.16.2 Builtin GET_DEVICE_REVISION

Communication profiles: CS, GPE, PB, PN

Purpose: Get DEVICE_REVISION from identification

The builtin GET_DEVICE_REVISION gets the DEVICE_REVISION from the currently used device description as stated in the identification of the device description with the keyword DEVICE_REVISION.

Lexical structure

GET_DEVICE_REVISION (VOID)

The lexical element of the builtin GET_DEVICE_REVISION is shown in Table 367.

Table 367 – Builtin GET_DEVICE_REVISION

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	The current DEVICE_REVISION.

6.16.3 Builtin GET_DEVICE_TYPE

Communication profiles: CS, GPE, PB, PN

Purpose: Get DEVICE_TYPE from identification

The builtin GET_DEVICE_TYPE gets the numerical equivalent of the DEVICE_TYPE from the currently used device description as stated in the identification of the device description with the keyword DEVICE_TYPE.

Lexical structure

GET_DEVICE_TYPE (VOID)

The lexical element of the builtin GET_DEVICE_TYPE is shown in Table 368.

Table 368 – Builtin GET_DEVICE_TYPE

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	The current DEVICE_TYPE.

6.16.4 Builtin GET_MANUFACTURER

Communication profiles: CS, GPE, PB, PN

Purpose: Get MANUFACTURER from identification

The builtin GET_MANUFACTURER gets the numerical equivalent of the MANUFACTURER from the currently used device description as stated in the identification of the device description with the keyword MANUFACTURER.

Lexical structure

GET_MANUFACTURER (VOID)

The lexical element of the builtin GET_MANUFACTURER is shown in Table 369.

Table 369 – Builtin GET_MANUFACTURER

Parameter name	Type	Direction	Usage	Description
<return>	int	O	m	The current MANUFACTURER.

6.17 Development support builtins**6.17.1 Builtin _ERROR**

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Logs error messages

The builtin _ERROR puts a textual information in a log. This information is put into the category error. The implementation is specific to the EDD application. Depending on the EDD application, the message may be piped to one or more files or even a diagnostic screen.

Any variable reference contained in the format string will be expanded to the current value. Method-variables and VARIABLES can be referenced in the message parameter. For more information, refer to 6.2.40.

In no case shall invocation of these functions result in the dispatching commands or other device communication.

Lexical structure

_ERROR(text)

The lexical elements of the builtin _ERROR are shown in Table 370.

Table 370 – Builtin _ERROR

Parameter name	Type	Direction	Usage	Description
text	DD_STRING	I	m	The textual information to be put into the log (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.17.2 Builtin _TRACE

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Logs trace point messages

The builtin _TRACE puts a textual information in a log. This information is put into the category trace. The implementation is specific to the EDD application. Depending on the EDD application, the message may be piped to one or more files or even a diagnostic screen.

Any variable reference contained in the format string will be expanded to the current value. Method-variables and VARIABLES can be referenced in the message parameter. For more information, refer to 6.2.40.

In no case shall invocation of these functions result in the dispatching commands or other device communication.

Lexical structure

_TRACE (text)

The lexical elements of the builtin _TRACE are shown in Table 371.

Table 371 – Builtin _TRACE

Parameter name	Type	Direction	Usage	Description
text	DD_STRING	I	m	The textual information to be put into the log (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.17.3 Builtin _WARNING

Communication profiles: CS, FF, GPE, HART, ISA100, PB, PN

Purpose: Logs warning messages

The builtin _WARNING puts a textual information in a log. This information is put into the category warning. The implementation is specific to the EDD application. Depending on the EDD application, the message may be piped to one or more files or even a diagnostic screen.

Any variable reference contained in the format string will be expanded to the current value. Method-variables and VARIABLES can be referenced in the message parameter. For more information, refer to 6.2.40.

In no case shall invocation of these functions result in the dispatching commands or other device communication.

Lexical structure

`_WARNING(text)`

The lexical elements of the builtin `_WARNING` are shown in Table 372.

Table 372 – Builtin `_WARNING`

Parameter name	Type	Direction	Usage	Description
text	DD_STRING	I	m	The textual information to be put into the log (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.17.4 Builtin LOG_MESSAGE

Communication profiles: CS, GPE, HART, PB, PN

Purpose: Log a message in the EDD application

The builtin LOG_MESSAGE inserts a message into a logging protocol. This builtin shall only be used as a result of a METHOD encountering a field device anomaly. This builtin shall not be used to assist in method debugging.

While implementation of the builtin LOG_MESSAGE is EDD application-specific, in general, the message parameter is used to forward a string to the EDD application's error manager. The message string may contain method-variable and/or VARIABLE values (see 6.2.40 for syntax). In no case shall invocation of these functions result in the dispatching commands or other device communication.

Lexical structure

`LOG_MESSAGE(priority, message)`

The lexical elements of the builtin LOG_MESSAGE are shown in Table 373.

Table 373 – Builtin LOG_MESSAGE

Parameter name	Type	Direction	Usage	Description
priority	int	I	m	The priority of the message (0 = error, 1 = warning, 2 = low).
message	DD_STRING	I	m	The message string to insert into the logging protocol (see 6.2.40 for syntax). String types that are not a DD_STRING may be also be used and will be automatically promoted to a DD_STRING (see 6.18.3).
<return>	VOID	—	—	—

6.18 DDL references

6.18.1 Direct referencing of VARIABLES

All EDD-VARIABLES are treated as external global variables from within a method. Any VARIABLE may be directly referenced from a METHOD DEFINITION in the same manner as a local automatic C-variable. For example:

```
ddVariable = ddVariable + 1
```

The legacy technique of accessing VARIABLES using library functions is still supported ensuring existing EDDs remain compliant and fully functional. In other words, referencing a VARIABLE using techniques like the following remains supported.

```
float loc_var;
loc_var = fvar_value(ddVariable) + 1;
fassign(ddVariable, loc_var);
```

VARIABLES may also be directly referenced in library functions like those supporting the user interface. For example,

```
result = SELECT_FROM_LIST("Units =#{pressure_units}", "YES;NO");
```

With direct VARIABLE referencing, the lower case version of functions like SELECT_FROM_LIST are no longer needed. Of course, to maintain backwards compatibility, the lower case functions are still supported. These lower case versions of user interface functions display a string and allow references to local variables and VARIABLES (referenced through an array of VARIABLE id's). For example, the lower case, select_from_list access a VARIABLE in the following manner:

```
long ids[1];
int result;
ids[0] = VARID(pressure_units);
result = select_from_list ("Units =#{0}", ids, "YES;NO");
```

Of course, direct VARIABLE references can also be used as the index value in an array. Consequently, the following syntax can be used:

```
GET_DEV_VAR_VALUE("Enter Units",
    dynamic_variables[analog_output_code].ANALOG_UNITS);
```

6.18.2 Accessing standard dictionary strings

Strings in the Standard Dictionary may be directly referenced using the dictionary_string builtin function. That means, any string literal like:

```
ACKNOWLEDGE ( "WARNING Remove Loop from automatic control" );
```

can be directly converted to using a string from the Standard Dictionary as follows;

```
ACKNOWLEDGE ( dictionary_string ( remove_loop_from_auto ) );
```

The legacy technique for accessing the Standard Dictionary using a character array to hold the string, and calling the method function get_dictionary_string, is still supported. For example, the following technique is still available:

```
char disp_string1[60]; /* an array to hold the string */
int slen1;
slen1 = 60;           /* the size of the string */

get_dictionary_string(remove_loop_from_auto, disp_string1, slen1);
ACKNOWLEDGE(disp_string1);
```

6.18.3 String support using DD_STRING

The method-variable type DD_STRING is an auto-sizing method-variable string type. Method variables of this type can be set using =, and concatenated using +. For example:

```
DD_STRING String1, String2, String3;
String1 = "This is fun";
String2= " isn't it?";
String3 = String1 + String2;
```

String3 would have the value of "This is fun, isn't it?".

NOTE 1 EDD developers need to carefully control the length of the DD_STRING they require. Some EDD applications (particularly the handhelds) can have memory constraints that limit available string space.

DD_STRINGs also support promotion. Many builtin functions in this specification list a DD_STRING as a parameter. In these cases, a DD_STRING or any string reference listed in IEC 61804-3 may be used. When string reference is the parameter, a temporary DD_STRING is created and passed to the method-function.

NOTE 2 A string reference promoted to a DD_STRING function parameter is not affected by modifications performed in the method-function.

A string VARIABLE (see IEC 61804-3) and a DD_STRING may be freely assigned on to the other. When a string VARIABLE (ASCII, PACKED_ASCII, PASSWORD) is assigned to a DD_STRING, the resulting string may be used to manipulate the value of that string variable. When a DD_STRING is assigned to a string VARIABLE some loss of information may occur. For example, a DD_STRING can represent a wider variety of character glyphs than possible when using a string VARIABLE of type PACKED_ASCII.

6.18.4 Referencing DD item attributes

Attributes of EDD Items may be directly referenced in EDD METHOD DEFINITIONS. The attribute that can be referenced depends on the type of the EDD Item, as illustrated in Table 374.

Table 374 – Referencable EDD item attributes

EDD item type	Type
EDD Item	Item.LABEL
	Item.HELP
VARIABLE	VARIABLE.DEFAULT_VALUE
Arithmetic VARIABLE	VARIABLE.MIN_VALUE
	VARIABLE.MAX_VALUE _n
	VARIABLE.MIN_VALUE
	VARIABLE.MAX_VALUE _n
AXIS	AXIS.VIEW_MIN
	AXIS.VIEW_MAX
LIST	LIST.CAPACITY
	LIST.COUNT
	LIST.FIRST
	LIST.LAST

6.19 Method functions

Methods are allowed to call other methods defined within the EDD (see IEC 61804-3). A METHOD can specify parameters and a return value:

```
METHOD identifier ( [methods-parameter-list]? )
{
    [TYPE method-type-specifier;]?
    ...
}
```

METHODS are called using the same syntax as calling a ANSI C function and may be used in the same way as an ANSI C-function. The invocation of a METHOD takes the form:

```
method-identifier ([methods-parameter-list]? );
```

where method-identifier is the identifier of another METHOD and the parameters are the arguments to be passed to the method-function. Parameters may be passed value or by reference and DD_STRING and DD_ITEM types are always passed by reference (see IEC 61804-3). The parameter-list shall not contain a variable number of arguments (i.e. the "..." notation is not supported). Recursive method-functions are supported.

The METHOD can also have a return value, specified by the TYPE attribute in the METHOD. All methods defined without a TYPE construct are assumed to return void.

METHODS and abort-methods run in the same environment as the original METHOD and use the same secondary data cache. Consequently, modifications made to a VARIABLE by any METHOD are available to all methods running in the same execution sequence. Method-variables are automatic ANSI C-variables and have the same scope as they would in an ANSI C-function.

Communication profiles: HART

Retry and abort masks, as well as retry count reset to default values at the start of each method running in the same execution sequence.

Any abort-methods added by a METHOD persist for the duration of the method-function's (and its subroutines) execution. The METHOD abort-methods are discarded when the METHOD returns to the calling METHOD.

Should an abort occur while the METHOD is executing, its abort-methods are executed followed by the execution of the calling METHOD's abort-methods (and then the abort-methods for its calling METHOD, etc.). Of course, VARIABLES remain global in scope, and, consequently, any changes to a VARIABLE remain in the secondary cache until all abort-methods have executed and the method execution environment is closed.

As an example of a method-function, consider a method "Multiply", that takes two integers as input and multiplies them together and returns an integer (see Figure 1):

```

METHOD Multiply(int A, int B)
{
    LABEL "Multiply";
    HELP "Multiplies integers";
    TYPE int;
    DEFINITION
    {
        return A*B;
    }
}

METHOD Caller
{
    DEFINITION
    {
        int A, B, C;
        A = 120;
        B = 2;
        C = Multiply(A, B);
    }
}

```

Figure 1 – A sample METHOD

The return value of a method can also be a string (see Figure 2):

```

METHOD Stringreverse(DD_STRING & string)
{
    TYPE DD_STRING;
    DEFINITION
    {
        DD_STRING reverse_string;
        int i, k;
        reverse_string = "";
        i = STRLEN(string);
        for ( k = 0; i; i--, k++ )
        {
            reverse_string[k] = string[i-1];
        }
        return reverse_string;
    }
}

METHOD Caller
{
    DEFINITION
    {
        send(20);

        // Note the conversion from DD-Variable long_tag to DD_STRING

        long_tag = Stringreverse ( long_tag );
        send(22);
    }
}

```

Figure 2 – DD_STRING parameters passed by reference

In Figure 2, a DD_STRING is a parameter to the METHOD Stringreverse and the VARIABLE long_tag is the designated parameter. If Stringreverse was to modify string, long_tag would be unaffected even though DD_STRING types are always passed by reference. This is because long_tag is promoted to a DD_STRING when Stringreverse is called (and a temporary DD_STRING is created in the process initialized with the value of long_tag).

7 Builtins return codes

Some builtins have specific return codes. These return codes are specified in Table 376 and Table 377. The description of Table 376 and Table 377 is in Table 375.

Table 375 – Contents of the return codes description table

Column	Meaning
Value	Specifies a long integer returned from the builtin.
Identifier	Specifies a reference to a returned long integer value.
Description	Specifies a description for the return code.

Table 376 – Return code descriptions

Value	Identifier	Description
0	BLTIN_SUCCESS	Success.
1801	BLTIN_NO_MEMORY	Out of memory.
1802	BLTIN_VAR_NOT_FOUND	Cannot find variable.
1803	BLTIN_BAD_ID	Specified item or member ID does not exist.
1804	BLTIN_NO_DATA_TO_SEND	No value to send to device.
1805	BLTIN_WRONG_DATA_TYPE	Wrong variable type for builtin.
1806	BLTIN_NO_RESP_CODES	Item does not have response codes.
1807	BLTIN_BAD_METHOD_ID	Invalid method ID.
1808	BLTIN_BUFFER_TOO_SMALL	Supplied buffer for starting string is too small.
1809	BLTIN_CANNOT_READ_VARIABLE	Cannot read the specified variable.
1810	BLTIN_INVALID_PROMPT	Bad prompt string.
1811	BLTIN_NO_LANGUAGE_STRING	No valid string for current language.
1812	BLTIN_DDS_ERROR	A DDS error occurred.
1813	BLTIN_FAIL_RESPONSE_CODE	Response code received from the device caused builtin to fail.
1814	BLTIN_FAIL_COMM_ERROR	Communication error caused builtin to fail.
1815	BLTIN_NOT_IMPLEMENTED	Builtin function not yet implemented.
1816	BLTIN_BAD_ITEM_TYPE	Wrong item type for builtin.
1817	BLTIN_VALUE_NOT_SET	A CLASS LOCAL variable has not been initialized before being used.
1818	BLTIN_BLOCK_NOT_FOUND	Specified block does not exist within the device.

Table 377 – Return code descriptions

Value	Identifier	Description
0	BI_SUCCESS	The builtin completed successfully.
-1	BI_ERROR	An error occurred during the execution of the builtin.
-2	BI_ABORT	The user aborted the task during the execution of the builtin.
-3	BI_NO_DEVICE	There was no response from the device during the execution of the builtin.
-4	BI_COMM_ERR	A communications error occurred during the execution of the builtin.
-5	BI_CONTINUE	The transfer has been started and is occurring.
-6	BI_RETRY	The same EDD-Item is already in the process of being transferred.
-7	BI_PORT_IN_USE	The port is in use (i.e. a transfer is in process).

SOMMAIRE

AVANT-PROPOS	298
INTRODUCTION.....	301
1 Domaine d'application	302
2 Références normatives	302
3 Termes, définitions, acronymes et abréviations	302
3.1 Termes et définitions	302
3.2 Acronymes et abréviations	302
4 Conventions pour les descriptions de builtin	303
5 Catégories de builtins	304
5.1 Vue d'ensemble	304
5.2 Builtins de l'interface utilisateur	314
5.3 Builtins de communication	315
5.4 Builtins d'action.....	317
6 Bibliothèque de builtin EDDL	320
6.1 Généralités	320
6.2 Builtins de l'interface utilisateur	321
6.2.1 Builtin ACKNOWLEDGE	321
6.2.2 Builtin acknowledge	321
6.2.3 Builtin DELAY	322
6.2.4 Builtin delay	322
6.2.5 Builtin delayfor	323
6.2.6 Builtin delayfor2	324
6.2.7 Builtin DISPLAY	326
6.2.8 Builtin display	327
6.2.9 Builtin display_bitenum	327
6.2.10 Builtin display_builtin_error	328
6.2.11 Builtin display_comm_error	329
6.2.12 Builtin display_comm_status	329
6.2.13 Builtin display_device_status	330
6.2.14 Builtin display_dynamics	330
6.2.15 Builtin display_dynamics2	331
6.2.16 Builtin display_message	333
6.2.17 Builtin display_message2	334
6.2.18 Builtin display_response_code	335
6.2.19 Builtin display_response_status	336
6.2.20 Builtin display_xmtr_status	337
6.2.21 Builtin edit_device_value	337
6.2.22 Builtin edit_device_value2	339
6.2.23 Builtin edit_local_value	341
6.2.24 Builtin edit_local_value2	342
6.2.25 Builtin get_acknowledgement	343
6.2.26 Builtin get_acknowledgement2	344
6.2.27 Builtin GET_DEV_VAR_VALUE	346
6.2.28 Builtin get_dev_var_value	346
6.2.29 Builtin GET_LOCAL_VAR_VALUE	347
6.2.30 Builtin get_local_var_value	348

6.2.31	Builtin Menu	349
6.2.32	Builtin MenuDisplay (version A)	350
6.2.33	Builtin MenuDisplay (version B)	351
6.2.34	Builtin PUT_MESSAGE.....	352
6.2.35	Builtin put_message	354
6.2.36	Builtin SELECT_FROM_LIST.....	355
6.2.37	Builtin select_from_list.....	356
6.2.38	Builtin select_from_menu.....	357
6.2.39	Builtin select_from_menu2.....	358
6.2.40	Formats de chaîne d'invite.....	361
6.3	Builtins de communication	361
6.3.1	Builtin abort_on_all_comm_errors.....	361
6.3.2	Builtin ABORT_ON_ALL_COMM_STATUS	362
6.3.3	Builtin ABORT_ON_ALL_DEVICE_STATUS	362
6.3.4	Builtin ABORT_ON_ALL_RESPONSE_CODES.....	363
6.3.5	Builtin abort_on_all_response_codes.....	364
6.3.6	Builtin ABORT_ON_COMM_ERROR.....	364
6.3.7	Builtin abort_on_comm_error.....	365
6.3.8	Builtin ABORT_ON_COMM_STATUS	365
6.3.9	Builtin ABORT_ON_DEVICE_STATUS	366
6.3.10	Builtin ABORT_ON_NO_DEVICE (déconseillé).....	367
6.3.11	Builtin ABORT_ON_RESPONSE_CODE.....	367
6.3.12	Builtin abort_on_response_code.....	369
6.3.13	Builtin ext_send_command	369
6.3.14	Builtin ext_send_command_trans	370
6.3.15	Builtin fail_on_all_comm_errors	371
6.3.16	Builtin fail_on_all_response_codes	372
6.3.17	Builtin fail_on_comm_error	373
6.3.18	Builtin fail_on_response_code	373
6.3.19	Builtin get_more_status	374
6.3.20	Builtin IGNORE_ALL_COMM_STATUS.....	375
6.3.21	Builtin IGNORE_ALL_DEVICE_STATUS.....	376
6.3.22	Builtin IGNORE_ALL_RESPONSE_CODES.....	376
6.3.23	Builtin IGNORE_COMM_ERROR.....	377
6.3.24	Builtin IGNORE_COMM_STATUS.....	377
6.3.25	Builtin IGNORE_DEVICE_STATUS.....	378
6.3.26	Builtin IGNORE_NO_DEVICE (déconseillé)	379
6.3.27	Builtin IGNORE_RESPONSE_CODE	379
6.3.28	Builtin read_value.....	380
6.3.29	Builtin read_value2.....	380
6.3.30	Builtin ReadCommand	381
6.3.31	Builtin retry_on_all_comm_errors.....	382
6.3.32	Builtin RETRY_ON_ALL_COMM_STATUS.....	382
6.3.33	Builtin RETRY_ON_ALL_DEVICE_STATUS.....	383
6.3.34	Builtin RETRY_ON_ALL_RESPONSE_CODES.....	384
6.3.35	Builtin retry_on_all_response_codes	384
6.3.36	Builtin RETRY_ON_COMM_ERROR.....	385
6.3.37	Builtin retry_on_comm_error.....	385
6.3.38	Builtin RETRY_ON_COMM_STATUS.....	386

6.3.39	Builtin RETRY_ON_DEVICE_STATUS.....	387
6.3.40	Builtin RETRY_ON_NO_DEVICE (déconseillé)	387
6.3.41	Builtin RETRY_ON_RESPONSE_CODE	388
6.3.42	Builtin retry_on_response_code.....	389
6.3.43	Builtin send.....	389
6.3.44	Builtin send_all_values	390
6.3.45	Builtin send_command.....	391
6.3.46	Builtin send_command_trans	391
6.3.47	Builtin send_trans	392
6.3.48	Builtin send_value	393
6.3.49	Builtin send_value2	394
6.3.50	Builtin SET_NUMBER_OF_RETRIES	395
6.3.51	Builtin WriteCommand	395
6.3.52	Builtin XMTR_ABORT_ON_ALL_COMM_STATUS.....	396
6.3.53	Builtin XMTR_ABORT_ON_ALL_DATA.....	396
6.3.54	Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS.....	397
6.3.55	Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES	398
6.3.56	Builtin XMTR_ABORT_ON_COMM_ERROR	398
6.3.57	Builtin XMTR_ABORT_ON_COMM_STATUS	399
6.3.58	Builtin XMTR_ABORT_ON_DATA.....	399
6.3.59	Builtin XMTR_ABORT_ON_DEVICE_STATUS.....	400
6.3.60	Builtin XMTR_ABORT_ON_NO_DEVICE (déconseillé)	401
6.3.61	Builtin XMTR_ABORT_ON_RESPONSE_CODE	401
6.3.62	Builtin XMTR_IGNORE_ALL_COMM_STATUS	402
6.3.63	Builtin XMTR_IGNORE_ALL_DATA	402
6.3.64	Builtin XMTR_IGNORE_ALL_DEVICE_STATUS	403
6.3.65	Builtin XMTR_IGNORE_ALL_RESPONSE_CODES	403
6.3.66	Builtin XMTR_IGNORE_COMM_ERROR	404
6.3.67	Builtin XMTR_IGNORE_COMM_STATUS	405
6.3.68	Builtin XMTR_IGNORE_DATA	405
6.3.69	Builtin XMTR_IGNORE_DEVICE_STATUS	406
6.3.70	Builtin XMTR_IGNORE_NO_DEVICE (déconseillé).....	406
6.3.71	Builtin XMTR_IGNORE_RESPONSE_CODE.....	407
6.3.72	Builtin XMTR_RETRY_ON_ALL_COMM_STATUS	408
6.3.73	Builtin XMTR_RETRY_ON_ALL_DATA	408
6.3.74	Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS	409
6.3.75	Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES	409
6.3.76	Builtin XMTR_RETRY_ON_COMM_ERROR	410
6.3.77	Builtin XMTR_RETRY_ON_COMM_STATUS	410
6.3.78	Builtin XMTR_RETRY_ON_DATA	411
6.3.79	Builtin XMTR_RETRY_ON_DEVICE_STATUS	412
6.3.80	Builtin XMTR_RETRY_ON_NO_DEVICE (déconseillé).....	412
6.3.81	Builtin XMTR_RETRY_ON_RESPONSE_CODE.....	413
6.4	Builtins d'action.....	413
6.4.1	Builtin fgetval.....	413
6.4.2	Builtin fsetval	414
6.4.3	Builtin get_date	414
6.4.4	Builtin get_double.....	415
6.4.5	Builtin get_float.....	416

6.4.6	Builtin get_signed	416
6.4.7	Builtin get_string.....	417
6.4.8	Builtin get_unsigned	418
6.4.9	Builtin igetval.....	418
6.4.10	Builtin isetval	419
6.4.11	Builtin isOffline	419
6.4.12	Builtin lgetval	420
6.4.13	Builtin lsetval	420
6.4.14	Builtin put_date	421
6.4.15	Builtin put_double.....	422
6.4.16	Builtin put_float.....	422
6.4.17	Builtin put_signed	423
6.4.18	Builtin put_string.....	424
6.4.19	Builtin put_unsigned	424
6.4.20	Builtin sgetval	425
6.4.21	Builtin ssetval	426
6.5	Builtins d'accès aux variables	426
6.5.1	Builtin assign	426
6.5.2	Builtin assign_double.....	427
6.5.3	Builtin assign_float	428
6.5.4	Builtin assign_int	428
6.5.5	Builtin assign_var	429
6.5.6	Builtin assign2	429
6.5.7	Builtin dassign	431
6.5.8	Builtin fassign	431
6.5.9	Builtin float_value	432
6.5.10	Builtin fvar_value	432
6.5.11	Builtin get_date_value	432
6.5.12	Builtin get_date_value2	433
6.5.13	Builtin get_double_value.....	434
6.5.14	Builtin get_double_value2.....	435
6.5.15	Builtin get_float_value	436
6.5.16	Builtin get_float_value2	437
6.5.17	Builtin get_signed_value	438
6.5.18	Builtin get_signed_value2.....	438
6.5.19	Builtin get_string_value	439
6.5.20	Builtin get_string_value2.....	440
6.5.21	Builtin get_unsigned_value	441
6.5.22	Builtin get_unsigned_value2	442
6.5.23	Builtin iassign	443
6.5.24	Builtin int_value	444
6.5.25	Builtin ivar_value	444
6.5.26	Builtin lassign	445
6.5.27	Builtin long_value	445
6.5.28	Builtin lvar_value	445
6.5.29	Builtin put_date_value	446
6.5.30	Builtin put_date_value2	447
6.5.31	Builtin put_double_value.....	448
6.5.32	Builtin put_double_value2.....	449

6.5.33	Builtin put_float_value	450
6.5.34	Builtin put_float_value2	451
6.5.35	Builtin put_signed_value	452
6.5.36	Builtin put_signed_value2	452
6.5.37	Builtin put_string_value	453
6.5.38	Builtin put_string_value2	454
6.5.39	Builtin put_unsigned_value	455
6.5.40	Builtin put_unsigned_value2	456
6.5.41	Builtin ret_double_value	458
6.5.42	Builtin ret_double_value2	458
6.5.43	Builtin ret_float_value	459
6.5.44	Builtin ret_float_value2	459
6.5.45	Builtin ret_signed_value	460
6.5.46	Builtin ret_signed_value2	460
6.5.47	Builtin ret_unsigned_value	461
6.5.48	Builtin ret_unsigned_value2	462
6.5.49	Builtin vassign	462
6.6	Builtins d'achèvement normal	463
6.6.1	Builtin discard_on_exit	463
6.6.2	Builtin save_on_exit	464
6.6.3	Builtin save_values	465
6.6.4	Builtin send_on_exit	465
6.7	Builtins d'achèvement anormal	466
6.7.1	Builtin abort	466
6.7.2	Builtin add_abort_method (version A)	467
6.7.3	Builtin add_abort_method (version B)	467
6.7.4	Builtin method_abort	468
6.7.5	Builtin pop_abort_method	469
6.7.6	Builtin process_abort	469
6.7.7	Builtin push_abort_method	470
6.7.8	Builtin remove_abort_method (version A)	470
6.7.9	Builtin remove_abort_method (version B)	471
6.7.10	Builtin remove_all_abort_methods	471
6.8	Builtins de chaîne	472
6.8.1	Builtin atof	472
6.8.2	Builtin atoi	472
6.8.3	Builtin BUILD_MESSAGE	473
6.8.4	Builtin dictionary_string	473
6.8.5	Builtin ftoa	474
6.8.6	Builtin get_comm_error_string	474
6.8.7	Builtin get_dictionary_string	475
6.8.8	Builtin get_enum_string	476
6.8.9	Builtin get_response_code_string	477
6.8.10	Builtin get_rspcode_string	478
6.8.11	Builtin get_rspcode_string_by_id	478
6.8.12	Builtin get_status_code_string (déconseillé)	479
6.8.13	Builtin get_status_string	479
6.8.14	Builtin get_stddict_string	480
6.8.15	Builtin get_variable_string	481

6.8.16	Builtin itoa (version A)	481
6.8.17	Builtin itoa (version B)	482
6.8.18	Builtin strleft	482
6.8.19	Builtin strcmp	483
6.8.20	Builtin strlen	483
6.8.21	Builtin strlwr	484
6.8.22	Builtin strmid	485
6.8.23	Builtin strright	485
6.8.24	Builtin strstr	486
6.8.25	Builtin strtrim	486
6.8.26	Builtin strupr	487
6.8.27	Builtin trunc	488
6.9	Builtins de date et heure	488
6.9.1	Builtin AddTime	488
6.9.2	Builtin DATE_AND_TIME_VALUE_to_string	489
6.9.3	Builtin Date_to_DayOfMonth	489
6.9.4	Builtin DATE_to_days	490
6.9.5	Builtin Date_to_Month	490
6.9.6	Builtin DATE_to_string	490
6.9.7	Builtin Date_To_Time	491
6.9.8	Builtin Date_to_Year	491
6.9.9	Builtin days_to_DATE	492
6.9.10	Builtin DELAY_TIME	492
6.9.11	Builtin DiffTime	493
6.9.12	Builtin From_DATE_AND_TIME_VALUE	493
6.9.13	Builtin From_TIME_VALUE	494
6.9.14	Builtin GetCurrentDate	494
6.9.15	Builtin GetCurrentDateAndTime	494
6.9.16	Builtin GetCurrentTime	495
6.9.17	Builtin GET_TICK_COUNT	495
6.9.18	Builtin Make_Time	496
6.9.19	Builtin seconds_to_TIME_VALUE	496
6.9.20	Builtin seconds_to_TIME_VALUE8	497
6.9.21	Builtin Time_To_Date	497
6.9.22	Builtin TIME_VALUE_to_Hour	498
6.9.23	Builtin TIME_VALUE_to_Minute	498
6.9.24	Builtin TIME_VALUE_to_Second	499
6.9.25	Builtin TIME_VALUE_to_seconds	499
6.9.26	Builtin TIME_VALUE_to_string	500
6.9.27	Builtin timet_to_string	501
6.9.28	Builtin timet_to_TIME_VALUE	501
6.9.29	Builtin timet_to_TIME_VALUE	502
6.9.30	Builtin timet_to_TIME_VALUE8	502
6.9.31	Builtin To_Date	503
6.9.32	Builtin To_Date_and_Time	503
6.9.33	Builtin To_Time	504
6.9.34	Builtin To_TIME_VALUE	504
6.9.35	Builtin To_TIME_VALUE8	505
6.10	Builtins de diagnostic	505

6.10.1	Builtin get_comm_error	505
6.10.2	Builtin get_dds_error	506
6.10.3	Builtin get_response_code	507
6.11	Builtins mathématiques	508
6.11.1	Builtin abs	508
6.11.2	Builtin acos	509
6.11.3	Builtin asin	509
6.11.4	Builtin atan	510
6.11.5	Builtin ByteToDouble	510
6.11.6	Builtin ByteToFloat	511
6.11.7	Builtin ByteToLong	511
6.11.8	Builtin ByteToShort	512
6.11.9	Builtin cbrt	512
6.11.10	Builtin ceil	512
6.11.11	Builtin cos	513
6.11.12	Builtin cosh	513
6.11.13	Builtin DoubleToByte	514
6.11.14	Builtin drand	514
6.11.15	Builtin dseed	515
6.11.16	Builtin exp	515
6.11.17	Builtin FloatToByte	516
6.11.18	Builtin floor	516
6.11.19	Builtin fmod	517
6.11.20	Builtin fpclassify	517
6.11.21	Builtin is_NaN	518
6.11.22	Builtin log	519
6.11.23	Builtin log10	519
6.11.24	Builtin log2	519
6.11.25	Builtin LongToByte	520
6.11.26	Builtin nan	520
6.11.27	Builtin NaN_value	521
6.11.28	Builtin nanf	522
6.11.29	Builtin pow	522
6.11.30	Builtin round	523
6.11.31	Builtin ShortToByte	523
6.11.32	Builtin sin	524
6.11.33	Builtin sinh	524
6.11.34	Builtin sqrt	524
6.11.35	Builtin tan	525
6.11.36	Builtin tanh	525
6.12	Builtins de liste	526
6.12.1	Builtin get_date_lelem	526
6.12.2	Builtin get_date_lelem2	527
6.12.3	Builtin get_double_lelem	528
6.12.4	Builtin get_double_lelem2	528
6.12.5	Builtin get_float_lelem	529
6.12.6	Builtin get_float_lelem2	530
6.12.7	Builtin get_signed_lelem	531
6.12.8	Builtin get_signed_lelem2	532

6.12.9	Builtin get_string_lelem	533
6.12.10	Builtin get_string_lelem2.....	533
6.12.11	Builtin get_unsigned_lelem	534
6.12.12	Builtin get_unsigned_lelem2	535
6.12.13	Builtin ListDeleteElementAt (version A).....	536
6.12.14	Builtin ListDeleteElementAt (version B).....	537
6.12.15	Builtin ListDeleteElementAt2.....	537
6.12.16	Builtin ListInsert (version A).....	538
6.12.17	Builtin ListInsert (version B).....	539
6.12.18	Builtin ListInsert2.....	539
6.13	Builtins de traduction	540
6.13.1	Builtin DICT_ID.....	540
6.13.2	Builtin get_block_instance_by_objet_index	541
6.13.3	Builtin get_block_instance_by_tag	542
6.13.4	Builtin get_block_instance_count.....	543
6.13.5	Builtin get_resolve_status.....	543
6.13.6	Builtin ITEM_ID	544
6.13.7	Builtin MEMBER_ID	544
6.13.8	Builtin ObjectReference	545
6.13.9	Builtin resolve_array_ref.....	546
6.13.10	Builtin resolve_array_ref2	547
6.13.11	Builtin resolve_block_ref.....	547
6.13.12	Builtin resolve_block_ref2.....	548
6.13.13	Builtin resolve_list_ref	549
6.13.14	Builtin resolve_local_ref.....	549
6.13.15	Builtin resolve_local_ref2.....	550
6.13.16	Builtin resolve_param_list_ref.....	551
6.13.17	Builtin resolve_param_ref	551
6.13.18	Builtin resolve_param_ref2	552
6.13.19	Builtin resolve_record_ref.....	553
6.13.20	Builtin resolve_record_ref2	553
6.13.21	Builtin VARID.....	554
6.14	Builtins de transfert de blocs.....	555
6.14.1	Builtin abortTransferPort.....	555
6.14.2	Builtin closeTransferPort.....	555
6.14.3	Builtin fGetByte	555
6.14.4	Builtin get_transfer_status	556
6.14.5	Builtin openTransferPort	557
6.14.6	Builtin readItemFromDevice.....	557
6.14.7	Builtin writeItemToDevice	558
6.15	Builtins de fichier	558
6.15.1	Builtin browseIdentity.....	558
6.15.2	Builtin re_read_file.....	559
6.15.3	Builtin re_write_file	559
6.16	Builtins d'identification	560
6.16.1	Builtin GET_DD_REVISION	560
6.16.2	Builtin GET_DEVICE_REVISION	560
6.16.3	Builtin GET_DEVICE_TYPE.....	561
6.16.4	Builtin GET_MANUFACTURER.....	561

6.17	Builtins d'aide au développement.....	561
6.17.1	Builtin _ERROR.....	561
6.17.2	Builtin _TRACE.....	562
6.17.3	Builtin _WARNING.....	563
6.17.4	Builtin LOG_MESSAGE	563
6.18	Références DDL	564
6.18.1	Référencement direct de VARIABLES	564
6.18.2	Accès à des chaînes du dictionnaire normalisé.....	565
6.18.3	Prise en charge de chaîne à l'aide de DD_STRING	565
6.18.4	Référencement d'attributs d'éléments DD	566
6.19	Fonctions de méthode.....	566
7	Codes de retour de builtins.....	568
Figure 1 – Exemple de METHOD		567
Figure 2 – Paramètres DD_STRING transmis par référence.....		568
Tableau 1 – Format du tableau de vue d'ensemble de builtin		303
Tableau 2 – Contenu du tableau de vue d'ensemble de builtin.....		303
Tableau 3 – Format des tableaux d'éléments lexicaux des builtins		304
Tableau 4 – Contenu du tableau d'éléments lexicaux.....		304
Tableau 5 – Vue d'ensemble de builtin.....		304
Tableau 6 – Usage des builtins		314
Tableau 7 – Builtins de l'interface utilisateur		314
Tableau 8 – Builtins de communication		315
Tableau 9 – Builtins d'action		317
Tableau 10 – Builtin ACKNOWLEDGE		321
Tableau 11 – Builtin acknowledge		322
Tableau 12 – Builtin DELAY.....		322
Tableau 13 – Builtin delay.....		323
Tableau 14 – Builtin delayfor		323
Tableau 15 – Builtin delayfor2.....		326
Tableau 16 – Builtin DISPLAY		327
Tableau 17 – Builtin display		327
Tableau 18 – Builtin display_bitenum.....		328
Tableau 19 – Builtin display_builtin_error.....		328
Tableau 20 – Builtin display_comm_error.....		329
Tableau 21 – Builtin display_comm_status.....		330
Tableau 22 – Builtin display_device_status		330
Tableau 23 – Builtin display_dynamics.....		331
Tableau 24 – Builtin display_dynamics2.....		332
Tableau 25 – Builtin display_message		333
Tableau 26 – Builtin display_message2		335
Tableau 27 – Builtin display_response_code.....		336
Tableau 28 – Builtin display_response_status.....		337

Tableau 29 – Builtin display_xmtr_status	337
Tableau 30 – Builtin edit_device_value	338
Tableau 31 – Builtin edit_device_value2	340
Tableau 32 – Builtin edit_local_value	341
Tableau 33 – Builtin edit_local_value2	343
Tableau 34 – Builtin get_acknowledgement	344
Tableau 35 – Builtin get_acknowledgement2	345
Tableau 36 – Builtin GET_DEV_VAR_VALUE	346
Tableau 37 – Builtin get_dev_var_value	347
Tableau 38 – Builtin GET_LOCAL_VAR_VALUE	348
Tableau 39 – Builtin get_local_var_value	348
Tableau 40 – Builtin Menu	349
Tableau 41 – Builtin MenuDisplay	351
Tableau 42 – Builtin MenuDisplay	352
Tableau 43 – Builtin PUT_MESSAGE	353
Tableau 44 – Builtin put_message	355
Tableau 45 – Builtin SELECT_FROM_LIST	356
Tableau 46 – Builtin select_from_list	357
Tableau 47 – Builtin select_from_menu	358
Tableau 48 – Builtin select_from_menu2	360
Tableau 49 – Options de format	361
Tableau 50 – Builtin abort_on_all_comm_errors	362
Tableau 51 – Builtin ABORT_ON_ALL_COMM_STATUS	362
Tableau 52 – Builtin ABORT_ON_ALL_DEVICE_STATUS	363
Tableau 53 – Builtin ABORT_ON_ALL_RESPONSE_CODES	364
Tableau 54 – Builtin abort_on_all_response_codes	364
Tableau 55 – Builtin ABORT_ON_COMM_ERROR	365
Tableau 56 – Builtin abort_on_comm_error	365
Tableau 57 – Builtin ABORT_ON_COMM_STATUS	366
Tableau 58 – Builtin ABORT_ON_DEVICE_STATUS	367
Tableau 59 – Builtin ABORT_ON_NO_DEVICE	367
Tableau 60 – Masques d'interruption et de relance disponibles	368
Tableau 61 – Builtin ABORT_ON_RESPONSE_CODE	368
Tableau 62 – Builtin abort_on_response_code	369
Tableau 63 – Builtin ext_send_command	370
Tableau 64 – Builtin ext_send_command_trans	371
Tableau 65 – Builtin fail_on_all_comm_errors	372
Tableau 66 – Builtin fail_on_all_response_codes	372
Tableau 67 – Builtin fail_on_comm_error	373
Tableau 68 – Builtin fail_on_response_code	374
Tableau 69 – Builtin get_more_status	375
Tableau 70 – Builtin IGNORE_ALL_COMM_STATUS	375
Tableau 71 – Builtin IGNORE_ALL_DEVICE_STATUS	376

Tableau 72 – Builtin IGNORE_ALL_RESPONSE_CODES	377
Tableau 73 – Builtin IGNORE_COMM_ERROR	377
Tableau 74 – Builtin IGNORE_COMM_STATUS	378
Tableau 75 – Builtin IGNORE_DEVICE_STATUS	378
Tableau 76 – Builtin IGNORE_NO_DEVICE	379
Tableau 77 – Builtin IGNORE_RESPONSE_CODE	380
Tableau 78 – Builtin read_value	380
Tableau 79 – Builtin read_value2	381
Tableau 80 – Builtin ReadCommand	382
Tableau 81 – Builtin retry_on_all_comm_errors	382
Tableau 82 – Builtin RETRY_ON_ALL_COMM_STATUS	383
Tableau 83 – Builtin RETRY_ON_ALL_DEVICE_STATUS	383
Tableau 84 – Builtin RETRY_ON_ALL_RESPONSE_CODES	384
Tableau 85 – Builtin retry_on_all_response_codes	385
Tableau 86 – Builtin RETRY_ON_COMM_ERROR	385
Tableau 87 – Builtin retry_on_comm_error	386
Tableau 88 – Builtin RETRY_ON_COMM_STATUS	386
Tableau 89 – Builtin RETRY_ON_DEVICE_STATUS	387
Tableau 90 – Builtin RETRY_ON_NO_DEVICE	388
Tableau 91 – Builtin RETRY_ON_RESPONSE_CODE	388
Tableau 92 – Builtin retry_on_response_code	389
Tableau 93 – Builtin send	390
Tableau 94 – Builtin send_all_values	391
Tableau 95 – Builtin send_command	391
Tableau 96 – Builtin send_command_trans	392
Tableau 97 – Builtin send_trans	393
Tableau 98 – Builtin send_value	393
Tableau 99 – Builtin send_value2	395
Tableau 100 – Builtin SET_NUMBER_OF_RETRIES	395
Tableau 101 – Builtin WriteCommand	396
Tableau 102 – Builtin XMTR_ABORT_ON_ALL_COMM_STATUS	396
Tableau 103 – Builtin XMTR_ABORT_ON_ALL_DATA	397
Tableau 104 – Builtin XMTR_ABORT_ON_ALL_DEVICE_STATUS	397
Tableau 105 – Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES	398
Tableau 106 – Builtin XMTR_ABORT_ON_COMM_ERROR	399
Tableau 107 – Builtin XMTR_ABORT_ON_COMM_STATUS	399
Tableau 108 – Builtin XMTR_ABORT_ON_DATA	400
Tableau 109 – Builtin XMTR_ABORT_ON_DEVICE_STATUS	400
Tableau 110 – Builtin XMTR_ABORT_ON_NO_DEVICE	401
Tableau 111 – Builtin XMTR_ABORT_ON_RESPONSE_CODE	402
Tableau 112 – Builtin XMTR_IGNORE_ALL_COMM_STATUS	402
Tableau 113 – Builtin XMTR_IGNORE_ALL_DATA	403
Tableau 114 – Builtin XMTR_IGNORE_ALL_DEVICE_STATUS	403

Tableau 115 – Builtin XMTR_IGNORE_ALL_RESPONSE_CODES	404
Tableau 116 – Builtin XMTR_IGNORE_COMM_ERROR	404
Tableau 117 – Builtin XMTR_IGNORE_COMM_STATUS	405
Tableau 118 – Builtin XMTR_IGNORE_DATA	406
Tableau 119 – Builtin XMTR_IGNORE_DEVICE_STATUS	406
Tableau 120 – Builtin XMTR_IGNORE_NO_DEVICE	407
Tableau 121 – Builtin XMTR_IGNORE_RESPONSE_CODE	407
Tableau 122 – Builtin XMTR_RETRY_ON_ALL_COMM_STATUS	408
Tableau 123 – Builtin XMTR_RETRY_ON_ALL_DATA	409
Tableau 124 – Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS	409
Tableau 125 – Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES	410
Tableau 126 – Builtin XMTR_RETRY_ON_COMM_ERROR	410
Tableau 127 – Builtin XMTR_RETRY_ON_COMM_STATUS	411
Tableau 128 – Builtin XMTR_RETRY_ON_DATA	411
Tableau 129 – Builtin XMTR_RETRY_ON_DEVICE_STATUS	412
Tableau 130 – Builtin XMTR_RETRY_ON_NO_DEVICE	413
Tableau 131 – Builtin XMTR_RETRY_ON_RESPONSE_CODE	413
Tableau 132 – Builtin fgetval	414
Tableau 133 – Builtin fsetval	414
Tableau 134 – Builtin get_date	415
Tableau 135 – Builtin get_double	415
Tableau 136 – Builtin get_float	416
Tableau 137 – Builtin get_signed	417
Tableau 138 – Builtin get_string	417
Tableau 139 – Builtin get_unsigned	418
Tableau 140 – Builtin igetval	419
Tableau 141 – Builtin isetval	419
Tableau 142 – Builtin isOffline	420
Tableau 143 – Builtin lgetval	420
Tableau 144 – Builtin lsetval	421
Tableau 145 – Builtin put_date	421
Tableau 146 – Builtin put_double	422
Tableau 147 – Builtin put_float	423
Tableau 148 – Builtin put_signed	423
Tableau 149 – Builtin put_string	424
Tableau 150 – Builtin put_unsigned	425
Tableau 151 – Builtin sgetval	425
Tableau 152 – Builtin ssetval	426
Tableau 153 – Builtin assign	427
Tableau 154 – Builtin assign_double	427
Tableau 155 – Builtin assign_float	428
Tableau 156 – Builtin assign_int	428
Tableau 157 – Builtin assign_var	429

Tableau 158 – Builtin assign2	430
Tableau 159 – Builtin dassign	431
Tableau 160 – Builtin fassign	431
Tableau 161 – Builtin float_value	432
Tableau 162 – Builtin fvar_value	432
Tableau 163 – Builtin get_date_value	433
Tableau 164 – Builtin get_date_value2	434
Tableau 165 – Builtin get_double_value	435
Tableau 166 – Builtin get_double_value2	436
Tableau 167 – Builtin get_float_value	436
Tableau 168 – Builtin get_float_value2	437
Tableau 169 – Builtin get_signed_value	438
Tableau 170 – Builtin get_signed_value2	439
Tableau 171 – Builtin get_string_value	440
Tableau 172 – Builtin get_string_value2	441
Tableau 173 – Builtin get_unsigned_value	442
Tableau 174 – Builtin get_unsigned_value2	443
Tableau 175 – Builtin iassign	444
Tableau 176 – Builtin int_value	444
Tableau 177 – Builtin ivar_value	444
Tableau 178 – Builtin lassign	445
Tableau 179 – Builtin long_value	445
Tableau 180 – Builtin lvar_value	446
Tableau 181 – Builtin put_date_value	446
Tableau 182 – Builtin put_date_value2	448
Tableau 183 – Builtin put_double_value	449
Tableau 184 – Builtin put_double_value2	450
Tableau 185 – Builtin put_float_value	450
Tableau 186 – Builtin put_float_value2	451
Tableau 187 – Builtin put_signed_value	452
Tableau 188 – Builtin put_signed_value2	453
Tableau 189 – Builtin put_string_value	454
Tableau 190 – Builtin put_string_value2	455
Tableau 191 – Builtin put_unsigned_value	456
Tableau 192 – Builtin put_unsigned_value2	457
Tableau 193 – Builtin ret_double_value	458
Tableau 194 – Builtin ret_double_value2	458
Tableau 195 – Builtin ret_float_value	459
Tableau 196 – Builtin ret_float_value2	460
Tableau 197 – Builtin ret_signed_value	460
Tableau 198 – Builtin ret_signed_value2	461
Tableau 199 – Builtin ret_unsigned_value	461
Tableau 200 – Builtin ret_unsigned_value2	462

Tableau 201 – Builtin vassign	463
Tableau 202 – Builtin discard_on_exit.....	464
Tableau 203 – Builtin save_on_exit.....	465
Tableau 204 – Builtin save_values.....	465
Tableau 205 – Builtin send_on_exit	466
Tableau 206 – Builtin abort	467
Tableau 207 – Builtin add_abort_method	467
Tableau 208 – Builtin add_abort_method	468
Tableau 209 – Builtin method_abort.....	469
Tableau 210 – Builtin pop_abort_method	469
Tableau 211 – Builtin process_abort	470
Tableau 212 – Builtin push_abort_method	470
Tableau 213 – Builtin remove_abort_method	471
Tableau 214 – Builtin remove_abort_method	471
Tableau 215 – Builtin remove_all_abort_methods	472
Tableau 216 – Builtin atof	472
Tableau 217 – Builtin atoi	473
Tableau 218 – Builtin BUILD_MESSAGE	473
Tableau 219 – Builtin dictionary_string	474
Tableau 220 – Builtin ftoa	474
Tableau 221 – Builtin get_comm_error_string	475
Tableau 222 – Builtin get_dictionary_string.....	476
Tableau 223 – Builtin get_enum_string	476
Tableau 224 – Builtin get_response_code_string	477
Tableau 225 – Builtin get_rspcode_string	478
Tableau 226 – Builtin get_rspcode_string_by_id	479
Tableau 227 – Builtin get_status_code_string	479
Tableau 228 – Builtin get_status_string	480
Tableau 229 – Builtin get_stddict_string.....	481
Tableau 230 – Builtin get_variable_string	481
Tableau 231 – Builtin itoa (version A)	482
Tableau 232 – Builtin itoa (version B)	482
Tableau 233 – Builtin strleft	483
Tableau 234 – Builtin strcmp.....	483
Tableau 235 – Builtin strlen	484
Tableau 236 – Builtin strlwr.....	484
Tableau 237 – Builtin strmid.....	485
Tableau 238 – Builtin strright	486
Tableau 239 – Builtin strstr	486
Tableau 240 – Builtin strtrim	487
Tableau 241 – Builtinstrupr	487
Tableau 242 – Builtin trunc	488
Tableau 243 – Builtin AddTime	488

Tableau 244 – Builtin DATE_AND_TIME_VALUE_to_string	489
Tableau 245 – Builtin Date_to_DayOfMonth.....	489
Tableau 246 – Builtin DATE_to_days	490
Tableau 247 – Builtin Date_to_Month	490
Tableau 248 – Builtin DATE_to_string.....	491
Tableau 249 – Builtin Date_To_Time	491
Tableau 250 – Builtin Date_to_Year.....	492
Tableau 251 – Builtin days_to_DATE	492
Tableau 252 – Builtin DELAY_TIME	492
Tableau 253 – Builtin DiffTime	493
Tableau 254 – Builtin From_DATE_AND_TIME_VALUE	493
Tableau 255 – Builtin From_TIME_VALUE	494
Tableau 256 – Builtin GetCurrentDate.....	494
Tableau 257 – Builtin GetCurrentDateAndTime	495
Tableau 258 – Builtin GetCurrentTime	495
Tableau 259 – Builtin GET_TICK_COUNT	496
Tableau 260 – Builtin Make_Time	496
Tableau 261 – Builtin seconds_to_TIME_VALUE	497
Tableau 262 – Builtin seconds_to_TIME_VALUE	497
Tableau 263 – Builtin Time_To_Date	497
Tableau 264 – Builtin TIME_VALUE_to_Hour.....	498
Tableau 265 – Builtin TIME_VALUE_to_Minute.....	498
Tableau 266 – Builtin TIME_VALUE_to_Second	499
Tableau 267 – Builtin TIME_VALUE_to_seconds	500
Tableau 268 – Builtin TIME_VALUE_to_string	500
Tableau 269 – Builtin timet_to_string	501
Tableau 270 – Builtin timet_to_TIME_VALUE	502
Tableau 271 – Builtin timet_To_TIME_VALUE	502
Tableau 272 – Builtin timet_to_TIME_VALUE8.....	502
Tableau 273 – Builtin To_Date.....	503
Tableau 274 – Builtin To_Date_and_Time.....	503
Tableau 275 – Builtin To_Time	504
Tableau 276 – Builtin To_TIME_VALUE	504
Tableau 277 – Builtin To_TIME_VALUE8	505
Tableau 278 – Builtin get_comm_error.....	506
Tableau 279 – Builtin get_ddb_error	507
Tableau 280 – Builtin get_response_code.....	508
Tableau 281 – Builtin abs	508
Tableau 282 – Builtin acos.....	509
Tableau 283 – Builtin asin.....	509
Tableau 284 – Builtin atan	510
Tableau 285 – Builtin ByteToDouble	510
Tableau 286 – Builtin ByteToFloat	511

Tableau 287 – Builtin ByteToLong	511
Tableau 288 – Builtin ByteToShort	512
Tableau 289 – Builtin cbrt	512
Tableau 290 – Builtin ceil	513
Tableau 291 – Builtin cos	513
Tableau 292 – Builtin cosh	514
Tableau 293 – Builtin DoubleToByte	514
Tableau 294 – Builtin drand	515
Tableau 295 – Builtin dseed	515
Tableau 296 – Builtin exp	516
Tableau 297 – Builtin FloatToByte	516
Tableau 298 – Builtin floor	517
Tableau 299 – Builtin fmod	517
Tableau 300 – Builtin fpclassify	518
Tableau 301 – Builtin is_NaN	518
Tableau 302 – Builtin log	519
Tableau 303 – Builtin log10	519
Tableau 304 – Builtin log2	520
Tableau 305 – Builtin LongToByte	520
Tableau 306 – Builtin nan	521
Tableau 307 – Builtin NaN_value	521
Tableau 308 – Builtin nanf	522
Tableau 309 – Builtin pow	523
Tableau 310 – Builtin round	523
Tableau 311 – Builtin ShortToByte	523
Tableau 312 – Builtin sin	524
Tableau 313 – Builtin sinh	524
Tableau 314 – Builtin sqrt	525
Tableau 315 – Builtin tan	525
Tableau 316 – Builtin tanh	526
Tableau 317 – Builtin get_date_lelem	526
Tableau 318 – Builtin get_date_lelem2	527
Tableau 319 – Builtin get_double_lelem	528
Tableau 320 – Builtin get_double_lelem2	529
Tableau 321 – Builtin get_float_lelem	530
Tableau 322 – Builtin get_float_lelem2	531
Tableau 323 – Builtin get_signed_lelem	531
Tableau 324 – Builtin get_signed_lelem2	532
Tableau 325 – Builtin get_string_lelem	533
Tableau 326 – Builtin get_string_lelem2	534
Tableau 327 – Builtin get_unsigned_lelem	535
Tableau 328 – Builtin get_unsigned_lelem2	536
Tableau 329 – Builtin ListDeleteElementAt	537

Tableau 330 – Builtin ListDeleteElementAt.....	537
Tableau 331 – Builtin ListDeleteElementAt2.....	538
Tableau 332 – Builtin ListInsert.....	539
Tableau 333 – Builtin ListInsert.....	539
Tableau 334 – Builtin ListInsert2.....	540
Tableau 335 – Builtin DICT_ID.....	541
Tableau 336 – Builtin get_block_instance_by_object_index	541
Tableau 337 – Builtin get_block_instance_by_tag	542
Tableau 338 – Builtin get_block_instance_count.....	543
Tableau 339 – Builtin get_resolve_status.....	544
Tableau 340 – Builtin ITEM_ID	544
Tableau 341 – Builtin MEMBER_ID.....	545
Tableau 342 – Builtin ObjectReference	545
Tableau 343 – Builtin resolve_array_ref.....	546
Tableau 344 – Builtin resolve_array_ref2.....	547
Tableau 345 – Builtin resolve_block_ref.....	548
Tableau 346 – Builtin resolve_block_ref2.....	549
Tableau 347 – Builtin resolve_list_ref	549
Tableau 348 – Builtin resolve_local_ref.....	550
Tableau 349 – Builtin resolve_local_ref2.....	550
Tableau 350 – Builtin resolve_param_list_ref.....	551
Tableau 351 – Builtin resolve_param_ref	552
Tableau 352 – Builtin resolve_param_ref2.....	552
Tableau 353 – Builtin resolve_record_ref	553
Tableau 354 – Builtin resolve_record_ref2.....	554
Tableau 355 – Builtin VARID.....	554
Tableau 356 – Builtin abortTransferPort.....	555
Tableau 357 – Builtin closeTransferPort.....	555
Tableau 358 – Builtin fGetByte.....	556
Tableau 359 – Builtin get_transfer_status	556
Tableau 360 – Builtin openTransferPort.....	557
Tableau 361 – Builtin readItemFromDevice.....	558
Tableau 362 – Builtin writeItemToDevice	558
Tableau 363 – Builtin browseIdentity.....	559
Tableau 364 – Builtin re_read_file.....	559
Tableau 365 – Builtin re_write_file	560
Tableau 366 – Builtin GET_DD_REVISION	560
Tableau 367 – Builtin GET_DEVICE_REVISION	560
Tableau 368 – Builtin GET_DEVICE_TYPE.....	561
Tableau 369 – Builtin GET_MANUFACTURER.....	561
Tableau 370 – Builtin _ERROR.....	562
Tableau 371 – Builtin _TRACE.....	562
Tableau 372 – Builtin _WARNING.....	563

Tableau 373 – Builtin LOG_MESSAGE	564
Tableau 374 – Attributs d'éléments EDD référençables	566
Tableau 375 – Contenu du tableau de descriptions de codes de retour	568
Tableau 376 – Descriptions de codes de retour	569
Tableau 377 – Descriptions de codes de retour	569

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

LES DISPOSITIFS ET LEUR INTÉGRATION DANS LES SYSTÈMES DE L'ENTREPRISE – BLOCS FONCTIONNELS (FB) POUR LES PROCÉDÉS INDUSTRIELS ET LE LANGAGE DE DESCRIPTION ÉLECTRONIQUE DE PRODUIT (EDDL) –

Partie 5: Bibliothèque de Builtin EDDL

AVANT-PROPOS

- 1) La Commission Electrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
- 2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
- 3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
- 4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
- 5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
- 6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
- 7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
- 8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
- 9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale IEC 61804-5 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Cette deuxième édition annule et remplace la première édition parue en 2015. Cette édition constitue une révision technique.

La présente édition a été élaborée en fusionnant le contenu de plusieurs variantes des spécifications de l'EDDL existantes, y compris celles du FieldComm Group (Foundation™ Fieldbus¹, HART®²), du PROFIBUS™³ Nutzerorganisation e.V. (PNO), et de l'ISA100_Wireless™⁴ Compliance Institute (ISA100 WCI). Par conséquent, le formatage et la numérotation de la présente édition peuvent différer des spécifications individuelles desquelles elle est issue.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

- ajout des profils de communication ISA100 et GPE;
- les builtins suivants ne sont plus conseillés:
 - ABORT_ON_NO_DEVICE
 - IGNORE_NO_DEVICE
 - RETRY_ON_NO_DEVICE
 - XMTR_ABORT_ON_NO_DEVICE
 - XMTR_IGNORE_NO_DEVICE
 - XMTR_RETRY_ON_NO_DEVICE
 - get_status_code_string

Le texte de cette Norme internationale est issu des documents suivants:

CDV	Rapport de vote
65E/634/CDV	65E/691/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette Norme internationale.

Cette publication a été rédigée selon les Directives ISO/IEC, Partie 2.

Une liste de toutes les parties de la série IEC 61804, publiées sous le titre général *Les dispositifs et leur intégration dans les systèmes de l'entreprise – Blocs fonctionnels (FB) pour les procédés industriels et le langage de description électronique de produit (EDDL)*, peut être consultée sur le site web de l'IEC.

Les futures normes de cette série porteront dorénavant le nouveau titre général cité ci-dessus. Le titre des normes existant déjà dans cette série sera mis à jour lors de la prochaine édition.

¹ FOUNDATION™ Fieldbus est l'appellation commerciale du FieldComm Group. Cette information est donnée à l'intention des utilisateurs du présent document et ne signifie nullement que l'IEC approuve ou recommande l'emploi exclusif du produit ainsi désigné. Des produits équivalents peuvent être utilisés s'il est démontré qu'ils conduisent aux mêmes résultats.

² HART® est une marque déposée du FieldComm Group. Cette information est donnée à l'intention des utilisateurs du présent document et ne signifie nullement que l'IEC approuve ou recommande l'emploi exclusif du produit ainsi désigné. Des produits équivalents peuvent être utilisés s'il est démontré qu'ils conduisent aux mêmes résultats.

³ PROFIBUS et PROFINET sont les appellations commerciales du PROFIBUS Nutzerorganisation e.V. Cette information est donnée à l'intention des utilisateurs du présent document et ne signifie nullement que l'IEC approuve ou recommande l'emploi exclusif du produit ainsi désigné. Des produits équivalents peuvent être utilisés s'il est démontré qu'ils conduisent aux mêmes résultats.

⁴ ISA100_Wireless™ est l'appellation commerciale de l'ISA100 Wireless Compliance Institute. Cette information est donnée à l'intention des utilisateurs du présent document et ne signifie nullement que l'IEC approuve ou recommande l'emploi exclusif du produit ainsi désigné. Des produits équivalents peuvent être utilisés s'il est démontré qu'ils conduisent aux mêmes résultats.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "<http://webstore.iec.ch>" dans les données relatives à la publication recherchée. A cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IMPORTANT – Le logo "*colour inside*" qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

INTRODUCTION

La présente partie de l'IEC 61804

- contient une liste des builtins qui peuvent être utilisés dans des méthodes EDDL;
- définit la fonctionnalité et les paramètres de chaque builtin;
- décrit comment des variables EDDL peuvent être référencées à partir de méthodes EDDL.

Les fonctions EDDL dépendent du profil de chacune des technologies de communication. Les descriptions que contient la présente partie de l'IEC 61804 renvoient à ces fonctions d'une manière générale et les technologies de communication ne prennent pas toutes en charge l'ensemble des fonctions décrites. Les définitions de profils indiquées dans la présentation de cette partie et dans l'IEC 61804-3 sont mentionnées de manière à comprendre les fonctions prises en charge par chaque technologie de communication.

IECNORM.COM : Click to view the full PDF of IEC 61804-5:2020

LES DISPOSITIFS ET LEUR INTÉGRATION DANS LES SYSTÈMES DE L'ENTREPRISE – BLOCS FONCTIONNELS (FB) POUR LES PROCÉDÉS INDUSTRIELS ET LE LANGAGE DE DESCRIPTION ÉLECTRONIQUE DE PRODUIT (EDDL) –

Partie 5: Bibliothèque de Builtin EDDL

1 Domaine d'application

La présente partie de l'IEC 61804 spécifie la bibliothèque de builtin EDDL et fournit les profils des différents bus de terrain.

2 Références normatives

Les documents suivants cités dans le texte constituent, pour tout ou partie de leur contenu, des exigences du présent document. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC 61804-3 :2020, *Les dispositifs et leur intégration dans les systèmes de l'entreprise – Blocs fonctionnels (FB) pour les procédés industriels et le langage de description électronique de produit (EDDL) – Partie 3: Sémantique et syntaxe EDDL*

ISO/IEC 9899, *Information technology – Programming Languages – C* (disponible en anglais seulement)

IEEE 754, *IEEE Standard for Binary Floating-Point Arithmetic* (disponible en anglais seulement)

3 Termes, définitions, acronymes et abréviations

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans l'IEC 61804-3 ainsi que les suivants s'appliquent.

L'ISO et l'IEC tiennent à jour des bases de données terminologiques destinées à être utilisées en normalisation, consultables aux adresses suivantes:

- IEC Electropedia: disponible à l'adresse <http://www.electropedia.org/>
- ISO Online browsing platform: disponible à l'adresse <http://www.iso.org/obp>

3.1.1

variable de méthode

variable à champ d'application dans une méthode

3.2 Acronymes et abréviations

CP	Communication Profile (Profil de communication)
CPF	Communication Profile Family (Famille de profils de communication)
CS	Communication Server (Serveur de communication)
DD	Device Description (Description de produit)

EDD	Electronic Device Description (Description électronique de produit)
EDDL	Electronic Device Description Language (Langage de description électronique de produit)
FF	Foundation Fieldbus
GPE	Generic Protocol Extension (Extension du protocole générique)
HART	HART® Communication Foundation
I/O	Input/output (Entrée/sortie)
ID	Identifiant
ISA100	ISA100_Wireless™
PB	PROFIBUS
PI	PROFIBUS & PROFINET ³ International
PN	PROFINET
PNO	PROFIBUS Nutzerorganisation

4 Conventions pour les descriptions de builtin

Les conventions suivantes s'appliquent pour la vue d'ensemble de builtin.

- Le Tableau 1 définit le format du tableau de vue d'ensemble de builtin.
- Le Tableau 2 définit le contenu des colonnes du Tableau 1.
- Les constructions et les attributs de base EDDL référencés dans le présent document sont écrits en majuscules.

Tableau 1 – Format du tableau de vue d'ensemble de builtin

Référence	Builtin	Profils de communication

Tableau 2 – Contenu du tableau de vue d'ensemble de builtin

Colonne	Texte	Signification
Référence		Référence au paragraphe qui spécifie le builtin.
Builtin		Nom du builtin.
Profils de communication	✓	Le builtin est pris en charge par le profil de communication correspondant.
	x	Le builtin n'est pas pris en charge par le profil de communication correspondant.
	d	L'utilisation du builtin n'est plus conseillée.

Les conventions suivantes s'appliquent pour les définitions de builtin.

- Le Tableau 3 définit le format des tableaux d'éléments lexicaux des builtins.
- Le Tableau 4 définit le contenu des colonnes du Tableau 3.
- Les constructions et les attributs de base EDDL référencés dans le présent document sont écrits en majuscules.

Tableau 3 – Format des tableaux d'éléments lexicaux des builtins

Nom du paramètre	Type	Direction	Utilisation	Description

Tableau 4 – Contenu du tableau d'éléments lexicaux

Colonne	Texte	Signification
Nom du paramètre	<name>	Nom du paramètre formel.
	"<return>"	Valeur de retour du builtin.
Type	data type	Type de données abstrait (de base ou dérivé) du paramètre formel.
Direction	I	Paramètre d'entrée.
	O	Un paramètre de sortie est transmis de manière implicite par la référence de sorte que la fonction builtin peut modifier la valeur. La valeur de retour d'un builtin est transmise par la valeur, le cas échéant.
	I/O	Un paramètre d'entrée/sortie est transmis de manière implicite par la référence de sorte que la fonction builtin peut modifier la valeur.
	—	Aucune valeur n'est retournée (VOID).
Utilisation	m	Le paramètre formel est obligatoire.
	o	Le paramètre formel est facultatif.
	c	Le paramètre formel est conditionnel.
	—	Aucune valeur n'est retournée (VOID).
Description	<text>	Texte descriptif pour le paramètre.
	—	Aucune valeur n'est retournée (VOID).

5 Catégories de builtins

5.1 Vue d'ensemble

Le Tableau 5 donne une présentation alphabétique de la bibliothèque de builtin EDDL et indique le profil par technologie de communication.

Tableau 5 – Vue d'ensemble de builtin

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.17.1	_ERROR	✓	✓	✓	✓	✓	✓	✓
6.17.2	_TRACE	✓	✓	✓	✓	✓	✓	✓
6.17.3	_WARNING	✓	✓	✓	✓	✓	✓	✓
6.7.1	abort	✓	✗	✓	✓	✗	✓	✓
6.3.1	abort_on_all_comm_errors	✗	✓	✗	✗	✓	✗	✗
6.3.2	ABORT_ON_ALL_COMM_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.3	ABORT_ON_ALL_DEVICE_STATUS	✗	✗	✗	✓	✗	✗	✗
6.3.4	ABORT_ON_ALL_RESPONSE_CODES	✗	✗	✗	✓	✗	✗	✗

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.5	abort_on_all_response_codes	x	✓	x	x	✓	x	x
6.3.6	ABORT_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.7	abort_on_comm_error	x	✓	x	x	✓	x	x
6.3.8	ABORT_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.9	ABORT_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.10	ABORT_ON_NO_DEVICE (déconseillé)	x	x	x	d	x	x	x
6.3.11	ABORT_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.12	abort_on_response_code	x	✓	x	x	✓	x	x
6.14.1	abortTransferPort	x	x	x	✓	x	x	x
6.11.1	abs	✓	✓	✓	✓	✓	✓	✓
6.2.1	ACKNOWLEDGE	✓	x	✓	✓	x	✓	✓
6.2.2	acknowledge	✓	x	✓	✓	x	✓	✓
6.11.2	acos	✓	✓	✓	✓	✓	✓	✓
6.7.2	add_abort_method (version A)	✓	x	✓	✓	x	✓	✓
6.7.3	add_abort_method (version B)	x	✓	x	x	✓	x	x
6.9.1	AddTime	✓	✓	✓	✓	✓	✓	✓
6.11.3	asin	✓	✓	✓	✓	✓	✓	✓
6.5.1	assign	x	✓	x	x	✓	x	x
6.5.2	assign_double	✓	x	✓	✓	x	✓	✓
6.5.3	assign_float	✓	x	✓	✓	x	✓	✓
6.5.4	assign_int	✓	x	✓	✓	x	✓	✓
6.5.5	assign_var	✓	x	✓	✓	x	✓	✓
6.5.6	assign2	x	✓	x	x	✓	x	x
6.11.4	atan	✓	✓	✓	✓	✓	✓	✓
6.8.1	atof	✓	x	✓	✓	x	✓	✓
6.8.2	atoi	✓	x	✓	✓	x	✓	✓
6.15.1	browseIdentity	x	x	x	✓	x	x	x
6.8.3	BUILD_MESSAGE	✓	✓	✓	✓	✓	✓	✓
6.11.5	ByteToDouble	✓	x	✓	x	x	✓	✓
6.11.6	ByteToFloat	✓	x	✓	x	x	✓	✓
6.11.7	ByteToLong	✓	x	✓	x	x	✓	✓
6.11.8	ByteToShort	✓	x	✓	x	x	✓	✓
6.11.9	cbrt	✓	✓	✓	✓	✓	✓	✓
6.11.10	ceil	✓	✓	✓	✓	✓	✓	✓
6.14.2	closeTransferPort	x	x	x	✓	x	x	x
6.11.11	cos	✓	✓	✓	✓	✓	✓	✓
6.11.12	cosh	✓	✓	✓	✓	✓	✓	✓
6.5.7	dassign	✓	x	✓	✓	x	✓	✓
6.9.2	DATE_AND_TIME_VALUE_to_string	✓	x	✓	✓	x	✓	✓
6.9.3	Date_to_DayOfMonth	✓	x	✓	✓	x	✓	✓

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.9.4	DATE_to_days	✓	✗	✓	✓	✗	✓	✓
6.9.5	Date_to_Month	✓	✗	✓	✓	✗	✓	✓
6.9.6	DATE_to_string	✓	✗	✓	✓	✗	✓	✓
6.9.7	Date_To_Time	✓	✗	✓	✓	✗	✓	✓
6.9.8	Date_to_Year	✓	✗	✓	✓	✗	✓	✓
6.9.9	days_to_DATE	✓	✗	✓	✓	✗	✓	✓
6.2.3	DELAY	✓	✗	✓	✓	✗	✓	✓
6.2.4	delay	✓	✗	✓	✓	✓	✓	✓
6.9.10	DELAY_TIME	✓	✗	✓	✓	✗	✓	✓
6.2.5	delayfor	✗	✓	✗	✗	✓	✗	✗
6.2.6	delayfor2	✗	✓	✗	✗	✓	✗	✗
6.13.1	DICT_ID	✗	✓	✗	✗	✓	✗	✗
6.8.4	dictionary_string	✓	✗	✓	✓	✗	✓	✓
6.9.11	DiffTime	✓	✓	✓	✓	✓	✓	✓
6.6.1	discard_on_exit	✓	✓	✓	✓	✓	✓	✓
6.2.7	DISPLAY	✓	✗	✓	✓	✗	✓	✓
6.2.8	display	✓	✗	✓	✓	✗	✓	✓
6.2.9	display_bitenum	✓	✗	✓	✗	✗	✓	✓
6.2.10	display_builtin_error	✗	✓	✗	✗	✓	✗	✗
6.2.11	display_comm_error	✗	✓	✗	✗	✓	✗	✗
6.2.12	display_comm_status	✗	✗	✗	✓	✗	✗	✗
6.2.13	display_device_status	✗	✗	✗	✓	✗	✗	✗
6.2.14	display_dynamics	✗	✓	✗	✗	✓	✗	✗
6.2.15	display_dynamics2	✗	✓	✗	✗	✓	✗	✗
6.2.16	display_message	✗	✓	✗	✗	✓	✗	✗
6.2.17	display_message2	✗	✓	✗	✗	✓	✗	✗
6.2.18	display_response_code	✗	✓	✗	✗	✓	✗	✗
6.2.19	display_response_status	✗	✗	✗	✓	✗	✗	✗
6.2.20	display_xmtr_status	✗	✗	✗	✓	✗	✗	✗
6.11.13	DoubleToByte	✓	✗	✓	✗	✗	✓	✓
6.11.14	drand	✓	✓	✓	✓	✓	✓	✓
6.11.15	dseed	✓	✓	✓	✓	✓	✓	✓
6.2.21	edit_device_value	✗	✓	✗	✗	✓	✗	✗
6.2.22	edit_device_value2	✗	✓	✗	✗	✓	✗	✗
6.2.23	edit_local_value	✗	✓	✗	✗	✓	✗	✗
6.2.24	edit_local_value2	✗	✓	✗	✗	✓	✗	✗
6.11.16	exp	✓	✓	✓	✓	✓	✓	✓
6.3.13	ext_send_command	✗	✗	✗	✓	✗	✗	✗
6.3.14	ext_send_command_trans	✗	✗	✗	✓	✗	✗	✗
6.3.15	fail_on_all_comm_errors	✗	✓	✗	✗	✓	✗	✗

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.16	fail_on_all_response_codes	x	✓	x	x	✓	x	x
6.3.17	fail_on_comm_error	x	✓	x	x	✓	x	x
6.3.18	fail_on_response_code	x	✓	x	x	✓	x	x
6.5.8	fassign	✓	x	✓	✓	x	✓	✓
6.14.2	fGetByte	x	x	x	✓	x	x	x
6.4.1	fgetval	✓	x	✓	✓	x		✓
6.5.9	float_value	✓	x	✓	✓	x		✓
6.11.17	FloatToByte	✓	x	✓	x		✓	✓
6.11.18	floor	✓	✓	✓	✓	✓	✓	✓
6.11.19	fmod	✓	✓	✓	✓	✓	✓	✓
6.11.20	fpclassify	✓	✓	✓	✓	✓	✓	✓
6.9.12	From_DATE_AND_TIME_VALUE	✓	x	✓	✓	x	✓	✓
6.9.13	From_TIME_VALUE	✓	x	✓	✓	x	✓	✓
6.4.2	fsetval	✓	x	✓	✓	x	✓	✓
6.8.5	ftoa	✓	x	✓	x	x	✓	✓
6.5.10	fvar_value	✓	x	✓	✓	x	✓	✓
6.2.25	get_acknowledgement	x	✓	x	x	✓	x	x
6.2.26	get_acknowledgement2	x	✓	x	x	✓	x	x
6.13.2	get_block_instance_by_object_index	x	✓	x	x	✓	x	x
6.13.3	get_block_instance_by_tag	x	✓	x	x	✓	x	x
6.13.4	get_block_instance_count	x	✓	x	x	✓	x	x
6.10.1	get_comm_error	x	✓	x	x	✓	x	x
6.8.6	get_comm_error_string	x	✓	x	x	✓	x	x
6.4.3	get_date	x	✓	x	x	✓	x	x
6.12.1	get_date_lelem	x	✓	x	x	✓	x	x
6.12.2	get_date_lelem2	x	✓	x	x	✓	x	x
6.5.11	get_date_value	x	✓	x	x	✓	x	x
6.5.12	get_date_value2	x	✓	x	x	✓	x	x
6.16.1	GET_DD_REVISION	✓	x	✓	x	x	✓	✓
6.10.2	get_dds_error	x	✓	x	x	✓	x	x
6.2.27	GET_DEV_VAR_VALUE	✓	x	✓	✓	x	✓	✓
6.2.28	get_dev_var_value	✓	x	✓	✓	x	✓	✓
6.16.2	GET_DEVICE_REVISION	✓	x	✓	x	x	✓	✓
6.16.3	GET_DEVICE_TYPE	✓	x	✓	x	x	✓	✓
6.8.7	get_dictionary_string	✓	x	✓	✓	x	✓	✓
6.4.4	get_double	x	✓	x	x	✓	x	x
6.12.3	get_double_lelem	x	✓	x	x	✓	x	x
6.12.4	get_double_lelem2	x	✓	x	x	✓	x	x
6.5.13	get_double_value	x	✓	x	x	✓	x	x
6.5.14	get_double_value2	x	✓	x	x	✓	x	x

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.8.8	get_enum_string	✓	x	✓	✓	x	✓	✓
6.4.5	get_float	x	✓	x	x	✓	x	x
6.12.5	get_float_lelem	x	✓	x	x	✓	x	x
6.12.6	get_float_lelem2	x	✓	x	x	✓	x	x
6.5.15	get_float_value	x	✓	x	x	✓	x	x
6.5.16	get_float_value2	x	✓	x	x	✓	x	x
6.2.29	GET_LOCAL_VAR_VALUE	✓	x	✓	✓	x	✓	✓
6.2.30	get_local_var_value	✓	x	✓	✓	✓	✓	✓
6.16.4	GET_MANUFACTURER	✓	x	✓	✓	x	✓	✓
6.3.19	get_more_status	x	x	x	x	x	x	x
6.13.5	get_resolve_status	x	✓	x	x	✓	x	x
6.10.3	get_response_code	x	✓	x	x	✓	x	x
6.8.9	get_response_code_string	x	✓	x	x	✓	x	x
6.8.10	get_rspcode_string	x	x	x	✓	x	x	x
6.8.11	get_rspcode_string_by_id	x	x	✓	x	x	✓	✓
6.4.6	get_signed	x	✓	x	x	✓	x	x
6.12.7	get_signed_lelem	x	✓	x	x	✓	x	x
6.12.8	get_signed_lelem2	x	✓	x	x	✓	x	x
6.5.17	get_signed_value	x	✓	x	x	✓	x	x
6.5.18	get_signed_value2	x	✓	x	x	✓	x	x
6.8.12	get_status_code_string (déconseillé)	x	x	x	d	x	x	x
6.8.13	get_status_string	x	✓	x	x	✓	x	x
6.8.14	get_stddict_string	x	✓	x	x	✓	x	x
6.4.7	get_string	x	✓	x	x	✓	x	x
6.12.9	get_string_lelem	x	✓	x	x	✓	x	x
6.12.10	get_string_lelem2	x	✓	x	x	✓	x	x
6.5.19	get_string_value	x	✓	x	x	✓	x	x
6.5.20	get_string_value2	x	✓	x	x	✓	x	x
6.9.17	GET_TICK_COUNT	✓	x	✓	x	x	✓	✓
6.14.4	get_transfer_status	x	x	x	✓	x	x	x
6.4.8	get_unsigned	x	✓	x	x	✓	x	x
6.12.11	get_unsigned_lelem	x	✓	x	x	✓	x	x
6.12.12	get_unsigned_lelem2	x	✓	x	x	✓	x	x
6.5.21	get_unsigned_value	x	✓	x	x	✓	x	x
6.5.22	get_unsigned_value2	x	✓	x	x	✓	x	x
6.8.15	get_variable_string	✓	x	✓	x	x	✓	✓
6.9.14	GetCurrentDate	✓	✓	✓	x	✓	✓	✓
6.9.15	GetCurrentDateAndTime	✓	✓	✓	x	✓	✓	✓
6.9.16	GetCurrentTime	✓	✓	✓	✓	✓	✓	✓
6.5.23	iassign	✓	x	✓	✓	x	✓	✓

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.4.9	igetval	✓	x	✓	✓	x	✓	✓
6.3.20	IGNORE_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.21	IGNORE_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.22	IGNORE_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.23	IGNORE_COMM_ERROR	x	x	x	✓	x	x	x
6.3.24	IGNORE_COMM_STATUS	x	x	x	✓	x	x	x
6.3.25	IGNORE_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.26	IGNORE_NO_DEVICE (déconseillé)	x	x	x	d	x	x	x
6.3.27	IGNORE_RESPONSE_CODE	x	x	x	✓	x	x	x
6.5.24	int_value	✓	x	✓	✓	x	✓	✓
6.11.21	is_NaN	✓	✓	✓	x	✓	✓	✓
6.4.10	isetval	✓	x	✓	✓	x	✓	✓
6.4.11	isOffline	✓	✓	✓	✓	✓	✓	✓
6.13.6	ITEM_ID	x	✓	x	x	✓	x	x
6.8.16	itoa (version A)	✓	x	✓	x	x	✓	✓
6.8.17	itoa (version B)	x	x	x	✓	x	x	x
6.5.25	ivar_value	✓	x	✓	✓	x	✓	✓
6.5.26	lassign	✓	x	✓	x	x	✓	✓
6.4.12	lgetval	✓	x	✓	x	x	✓	✓
6.12.13	ListDeleteElementAt (version A)	✓	x	✓	✓	x	✓	✓
6.12.14	ListDeleteElementAt (version B)	x	✓	x	x	✓	x	x
6.12.15	ListDeleteElementAt2	x	✓	x	x	✓	x	x
6.12.16	ListInsert (version A)	✓	x	✓	✓	x	✓	✓
6.12.17	ListInsert (version B)	x	✓	x	x	✓	x	x
6.12.18	ListInsert2	x	✓	x	x	✓	x	x
6.11.22	log	✓	✓	✓	✓	✓	✓	✓
6.17.4	LOG_MESSAGE	✓	x	✓	✓	x	✓	✓
6.11.23	log10	✓	✓	✓	✓	✓	✓	✓
6.11.24	log2	✓	✓	✓	✓	✓	✓	✓
6.5.27	long_value	✓	x	✓	x	x	✓	✓
6.11.25	LongToByte	✓	x	✓	x	x	✓	✓
6.4.13	lsetval	✓	x	✓	x	x	✓	✓
6.5.28	lvar_value	✓	x	✓	x	x	✓	✓
6.9.18	Make_Time	✓	✓	✓	✓	✓	✓	✓
6.13.7	MEMBER_ID	x	✓	x	x	✓	x	x
6.2.31	Menu	✓	x	✓	✓	x	✓	✓
6.2.32	MenuDisplay (version A)	✓	x	✓	✓	x	✓	✓
6.2.33	MenuDisplay (version B)	x	✓	x	x	✓	x	x
6.7.4	method_abort	x	✓	x	x	✓	x	x
6.11.26	nan	✓	✓	✓	✓	✓	✓	✓

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.11.27	NaN_value	✓	✓	✓	✗	✓	✓	✓
6.11.28	nanf	✓	✓	✓	✓	✓	✓	✓
6.13.8	ObjectReference	✓	✗	✓	✗	✗	✓	✓
6.14.5	openTransferPort	✗	✗	✗	✓	✗	✗	✗
6.7.5	pop_abort_method	✓	✗	✓	✓	✗	✓	✓
6.11.29	pow	✓	✓	✓	✓	✓	✓	✓
6.7.6	process_abort	✓	✗	✓	✓	✗	✓	✓
6.7.7	push_abort_method	✓	✗	✓	✓	✓	✓	✓
6.4.14	put_date	✗	✓	✗	✗	✓	✗	✗
6.5.29	put_date_value	✗	✓	✗	✗	✓	✗	✗
6.5.30	put_date_value2	✗	✓	✗	✗	✓	✗	✗
6.4.15	put_double	✗	✓	✗	✗	✓	✗	✗
6.5.31	put_double_value	✗	✓	✗	✗	✓	✗	✗
6.5.32	put_double_value2	✗	✓	✗	✗	✓	✗	✗
6.4.16	put_float	✗	✓	✗	✗	✓	✗	✗
6.5.33	put_float_value	✗	✓	✗	✗	✓	✗	✗
6.5.34	put_float_value2	✗	✓	✗	✗	✓	✗	✗
6.2.34	PUT_MESSAGE	✓	✗	✓	✓	✗	✓	✓
6.2.35	put_message	✓	✗	✓	✓	✗	✓	✓
6.4.17	put_signed	✗	✓	✗	✗	✓	✗	✗
6.5.35	put_signed_value	✗	✓	✗	✗	✓	✗	✗
6.5.36	put_signed_value2	✗	✓	✗	✗	✓	✗	✗
6.4.18	put_string	✗	✓	✗	✗	✓	✗	✗
6.5.37	put_string_value	✗	✓	✗	✗	✓	✗	✗
6.5.38	put_string_value2	✗	✓	✗	✗	✓	✗	✗
6.4.19	put_unsigned	✗	✓	✗	✗	✓	✗	✗
6.5.39	put_unsigned_value	✗	✓	✗	✗	✓	✗	✗
6.5.40	put_unsigned_value2	✗	✓	✗	✗	✓	✗	✗
6.15.2	re_read_file	✗	✗	✗	✓	✗	✗	✗
6.15.3	re_write_file	✗	✗	✗	✓	✗	✗	✗
6.3.28	read_value	✗	✓	✗	✗	✓	✗	✗
6.3.29	read_value2	✗	✓	✗	✗	✓	✗	✗
6.3.30	ReadCommand	✓	✗	✓	✗	✗	✓	✓
6.14.6	readItemFromDevice	✗	✗	✗	✓	✗	✗	✗
6.7.8	remove_abort_method (version A)	✓	✗	✓	✓	✗	✓	✓
6.7.9	remove_abort_method (version B)	✗	✓	✗	✗	✓	✗	✗
6.7.10	remove_all_abort_methods	✓	✓	✓	✓	✓	✓	✓
6.13.9	resolve_array_ref	✗	✓	✗	✗	✓	✗	✗
6.13.10	resolve_array_ref2	✗	✓	✗	✗	✓	✗	✗
6.13.11	resolve_block_ref	✗	✓	✗	✗	✓	✗	✗

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.13.12	resolve_block_ref2	x	✓	x	x	✓	x	x
6.13.13	resolve_list_ref	x	✓	x	x	✓	x	x
6.13.14	resolve_local_ref	x	✓	x	x	✓	x	x
6.13.15	resolve_local_ref2	x	✓	x	x	✓	x	x
6.13.16	resolve_param_list_ref	x	✓	x	x	✓	x	x
6.13.17	resolve_param_ref	x	✓	x	x	✓	x	x
6.13.18	resolve_param_ref2	x	✓	x	x	✓	x	x
6.13.19	resolve_record_ref	x	✓	x	x	✓	x	x
6.13.20	resolve_record_ref2	x	✓	x	x	✓	x	x
6.5.41	ret_double_value	x	✓	x	x	✓	x	x
6.5.42	ret_double_value2	x	✓	x	x	✓	x	x
6.5.43	ret_float_value	x	✓	x	x	✓	x	x
6.5.44	ret_float_value2	x	✓	x	x	✓	x	x
6.5.45	ret_signed_value	x	✓	x	x	✓	x	x
6.5.46	ret_signed_value2	x	✓	x	x	✓	x	x
6.5.47	ret_unsigned_value	x	✓	x	x	✓	x	x
6.5.48	ret_unsigned_value2	x	✓	x	x	✓	x	x
6.3.31	retry_on_all_comm_errors	x	✓	x	x	✓	x	x
6.3.32	RETRY_ON_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.33	RETRY_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.34	RETRY_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.35	retry_on_all_response_codes	x	✓	x	x	✓	x	x
6.3.36	RETRY_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.37	retry_on_comm_error	x	✓	x	x	✓	x	x
6.3.38	RETRY_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.39	RETRY_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.40	RETRY_ON_NO_DEVICE (déconseillé)	d	x	d	d	x	d	d
6.3.41	RETRY_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.42	retry_on_response_code	x	✓	x	x	✓	x	x
6.11.30	round	✓	✓	✓	✓	✓	✓	✓
6.6.2	save_on_exit	✓	✓	✓	x	✓	✓	✓
6.6.3	save_values	✓	x	✓	✓	x	✓	✓
6.9.19	seconds_to_TIME_VALUE	✓	x	✓	✓	x	✓	✓
6.9.20	seconds_to_TIME_VALUE8	✓	✓	✓	x	✓	✓	✓
6.2.36	SELECT_FROM_LIST	✓	x	✓	✓	x	✓	✓
6.2.37	select_from_list	✓	x	✓	✓	x	✓	✓
6.2.38	select_from_menu	x	✓	x	x	✓	x	x
6.2.39	select_from_menu2	x	✓	x	x	✓	x	x
6.3.43	send	x	x	x	✓	x	x	x
6.3.44	send_all_values	x	✓	x	x	✓	x	x

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.3.45	send_command	x	x	x	✓	x	x	x
6.3.46	send_command_trans	x	x	x	✓	x	x	x
6.6.4	send_on_exit	✓	✓	✓	x	✓	✓	✓
6.3.47	send_trans	x	x	x	✓	x	x	x
6.3.48	send_value	x	✓	x	x	✓	x	x
6.3.49	send_value2	x	✓	x	x	✓	x	x
6.3.50	SET_NUMBER_OF_RETRIES	✓	x	✓	✓	x	✓	✓
6.4.20	sgetval	✓	x	✓	✓	✓	✓	✓
6.11.31	ShortToByte	✓	x	✓	x	x	✓	✓
6.11.32	sin	✓	✓	✓	✓	✓	✓	✓
6.11.33	sinh	✓	✓	✓	✓	✓	✓	✓
6.11.34	sqrt	✓	✓	✓	✓	✓	✓	✓
6.4.21	ssetval	✓	x	✓	✓	x	✓	✓
6.8.19	strcmp	✓	✓	✓	✓	✓	✓	✓
6.8.18	strleft	✓	✓	✓	x	✓	✓	✓
6.8.20	strlen	✓	✓	✓	✓	✓	✓	✓
6.8.21	strlwr	✓	✓	✓	✓	✓	✓	✓
6.8.22	strmid	✓	✓	✓	✓	✓	✓	✓
6.8.23	strright	✓	✓	✓	x	✓	✓	✓
6.8.24	strstr	✓	✓	✓	✓	✓	✓	✓
6.8.25	strtrim	✓	✓	✓	✓	✓	✓	✓
6.8.26	strupr	✓	✓	✓	✓	✓	✓	✓
6.11.35	tan	✓	✓	✓	✓	✓	✓	✓
6.11.36	tanh	✓	✓	✓	✓	✓	✓	✓
6.9.21	Time_To_Date	✓	x	✓	✓	x	✓	✓
6.9.22	TIME_VALUE_to_Hour	✓	✓	✓	✓	✓	✓	✓
6.9.23	TIME_VALUE_to_Minute	✓	✓	✓	✓	✓	✓	✓
6.9.24	TIME_VALUE_to_Second	✓	✓	✓	✓	✓	✓	✓
6.9.25	TIME_VALUE_to_seconds	✓	✓	✓	✓	✓	✓	✓
6.9.26	TIME_VALUE_to_string	✓	✓	✓	✓	✓	✓	✓
6.9.27	timet_to_string	✓	✓	✓	✓	✓	✓	✓
6.9.28	timet_to_TIME_VALUE	✓	x	✓	x	x	✓	✓
6.9.29	timet_To_TIME_VALUE	x	x	x	✓	x	x	x
6.9.30	timet_to_TIME_VALUE8	✓	✓	✓	x	✓	✓	✓
6.9.31	To_Date	✓	x	✓	✓	x	✓	✓
6.9.32	To_Date_and_Time	✓	x	✓	x	x	✓	✓
6.9.33	To_Time	✓	x	✓	✓	x	✓	✓
6.9.34	To_TIME_VALUE	✓	x	✓	✓	x	✓	✓
6.9.35	To_TIME_VALUE8	✓	✓	✓	x	✓	✓	✓
6.8.27	trunc	✓	✓	✓	✓	✓	✓	✓

Référence	Builtin	Profils de communication						
		CS	FF	GPE	HART	ISA100	PB	PN
6.13.21	VARID	✓	x	✓	✓	x	✓	✓
6.5.49	vassign	✓	x	✓	✓	x	✓	✓
6.3.51	WriteCommand	✓	x	✓	x	x	✓	✓
6.14.7	writeltemToDevice	x	x	x	✓	x	x	x
6.3.52	XMTR_ABORT_ON_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.53	XMTR_ABORT_ON_ALL_DATA	x	x	x	✓	x	x	x
6.3.54	XMTR_ABORT_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.55	XMTR_ABORT_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.56	XMTR_ABORT_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.57	XMTR_ABORT_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.58	XMTR_ABORT_ON_DATA	x	x	x	✓	x	x	x
6.3.59	XMTR_ABORT_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.60	XMTR_ABORT_ON_NO_DEVICE (déconseillé)	x	x	x	d	x	x	x
6.3.61	XMTR_ABORT_ON_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.62	XMTR_IGNORE_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.63	XMTR_IGNORE_ALL_DATA	x	x	x	✓	x	x	x
6.3.64	XMTR_IGNORE_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.65	XMTR_IGNORE_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.66	XMTR_IGNORE_COMM_ERROR	x	x	x	✓	x	x	x
6.3.67	XMTR_IGNORE_COMM_STATUS	x	x	x	✓	x	x	x
6.3.68	XMTR_IGNORE_DATA	x	x	x	✓	x	x	x
6.3.69	XMTR_IGNORE_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.70	XMTR_IGNORE_NO_DEVICE (déconseillé)	x	x	x	d	x	x	x
6.3.71	XMTR_IGNORE_RESPONSE_CODE	x	x	x	✓	x	x	x
6.3.72	XMTR_RETRY_ON_ALL_COMM_STATUS	x	x	x	✓	x	x	x
6.3.73	XMTR_RETRY_ON_ALL_DATA	x	x	x	✓	x	x	x
6.3.74	XMTR_RETRY_ON_ALL_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.75	XMTR_RETRY_ON_ALL_RESPONSE_CODES	x	x	x	✓	x	x	x
6.3.76	XMTR_RETRY_ON_COMM_ERROR	x	x	x	✓	x	x	x
6.3.77	XMTR_RETRY_ON_COMM_STATUS	x	x	x	✓	x	x	x
6.3.78	XMTR_RETRY_ON_DATA	x	x	x	✓	x	x	x
6.3.79	XMTR_RETRY_ON_DEVICE_STATUS	x	x	x	✓	x	x	x
6.3.80	XMTR_RETRY_ON_NO_DEVICE (déconseillé)	x	x	x	d	x	x	x
6.3.81	XMTR_RETRY_ON_RESPONSE_CODE	x	x	x	✓	x	x	x

Les catégories de builtins spéciales sont les suivantes:

- builtins de l'interface utilisateur;
- builtins de communication;
- builtins d'action.

Le Tableau 6 présente les restrictions qui s'appliquent à l'usage des builtins, selon leur catégorie.

Tableau 6 – Usage des builtins

Usage des builtins	Builtins de l'interface utilisateur	Builtins de communication	Builtins d'action
Méthodes référencées dans les menus de l'interface utilisateur	Admis	Admis	Non admis
PRE_EDIT_ACTIONS	Admis	Non recommandés	Admis ^a
POST_EDIT_ACTIONS	Admis	Non recommandés	Admis ^a
REFRESH_ACTIONS	Non recommandés	GRAPH et CHART: admis Autres: non recommandés	Admis ^a
PRE_READ_ACTIONS	Non recommandés	Admis	Admis ^a
POST_READ_ACTIONS	Non recommandés	Admis	Admis ^a
PRE_WRITE_ACTIONS	Non recommandés	Admis	Admis ^a
POST_WRITE_ACTIONS	Non recommandés	Admis	Admis ^a
INIT_ACTIONS	Non recommandés	Admis	Non admis
EXIT_ACTIONS	Non recommandés	Admis	Non admis
^a Doivent être utilisés uniquement pour les actions appelées par une VARIABLE.			

5.2 Builtins de l'interface utilisateur

Les builtins de l'interface utilisateur affichent les informations et certains d'entre eux exigent en outre une interaction de l'utilisateur. Le Tableau 7 donne la liste des builtins de l'interface utilisateur.

Tableau 7 – Builtins de l'interface utilisateur

Nom du builtin
ACKNOWLEDGE
acknowledge
DELAY
delay
delayfor
delayfor2
DISPLAY
display
display_bitenum
display_builtin_error
display_comm_error
display_comm_status
display_device_status
display_dynamics
display_dynamics2
display_message
display_message2

Nom du builtin
display_response_code
display_response_status
display_xmtr_status
edit_device_value
edit_device_value2
edit_local_value
edit_local_value2
get_acknowledgement2
get_acknowledgement
GET_DEV_VAR_VALUE
get_dev_var_value
GET_LOCAL_VAR_VALUE
get_local_var_value
Menu
MenuDisplay (version A)
MenuDisplay (version B)
PUT_MESSAGE
put_message
SELECT_FROM_LIST
select_from_list
select_from_menu
select_from_menu2

5.3 Builtins de communication

Les builtins de communication permettent aux méthodes de communiquer avec le dispositif ou de contrôler la communication avec ce dernier. Le Tableau 8 donne la liste des builtins de communication.

Tableau 8 – Builtins de communication

Nom du builtin
abort_on_all_comm_errors
ABORT_ON_ALL_COMM_STATUS
ABORT_ON_ALL_DEVICE_STATUS
ABORT_ON_ALL_RESPONSE_CODES
abort_on_all_response_codes
ABORT_ON_COMM_ERROR
abort_on_comm_error
ABORT_ON_COMM_STATUS
ABORT_ON_DEVICE_STATUS
ABORT_ON_NO_DEVICE
ABORT_ON_RESPONSE_CODE
abort_on_response_code
ext_send_command
ext_send_command_trans

Nom du builtin
fail_on_all_comm_errors
fail_on_all_response_codes
fail_on_comm_error
fail_on_response_code
get_more_status
IGNORE_ALL_COMM_STATUS
IGNORE_ALL_DEVICE_STATUS
IGNORE_ALL_RESPONSE_CODES
IGNORE_COMM_ERROR
IGNORE_COMM_STATUS
IGNORE_DEVICE_STATUS
IGNORE_NO_DEVICE
IGNORE_RESPONSE_CODE
read_value
read_value2
ReadCommand
retry_on_all_comm_errors
RETRY_ON_ALL_COMM_STATUS
RETRY_ON_ALL_DEVICE_STATUS
RETRY_ON_ALL_RESPONSE_CODES
retry_on_all_response_codes
RETRY_ON_COMM_ERROR
retry_on_comm_error
RETRY_ON_COMM_STATUS
RETRY_ON_DEVICE_STATUS
RETRY_ON_NO_DEVICE
RETRY_ON_RESPONSE_CODE
retry_on_response_code
send
send_all_values
send_command
send_command_trans
send_trans
send_value
send_value2
SET_NUMBER_OF_RETRIES
WriteCommand
XMTR_ABORT_ON_ALL_COMM_STATUS
XMTR_ABORT_ON_ALL_DATA
XMTR_ABORT_ON_ALL_DEVICE_STATUS
XMTR_ABORT_ON_ALL_RESPONSE_CODES
XMTR_ABORT_ON_COMM_ERROR
XMTR_ABORT_ON_COMM_STATUS
XMTR_ABORT_ON_DATA

Nom du builtin
XMTR_ABORT_ON_DEVICE_STATUS
XMTR_ABORT_ON_NO_DEVICE
XMTR_ABORT_ON_RESPONSE_CODE
XMTR_IGNORE_ALL_COMM_STATUS
XMTR_IGNORE_ALL_DATA
XMTR_IGNORE_ALL_DEVICE_STATUS
XMTR_IGNORE_ALL_RESPONSE_CODES
XMTR_IGNORE_COMM_ERROR
XMTR_IGNORE_COMM_STATUS
XMTR_IGNORE_DATA
XMTR_IGNORE_DEVICE_STATUS
XMTR_IGNORE_NO_DEVICE
XMTR_IGNORE_RESPONSE_CODE
XMTR_RETRY_ON_ALL_COMM_STATUS
XMTR_RETRY_ON_ALL_DATA
XMTR_RETRY_ON_ALL_DEVICE_STATUS
XMTR_RETRY_ON_ALL_RESPONSE_CODE
XMTR_RETRY_ON_ALL_RESPONSE_CODES
XMTR_RETRY_ON_COMM_ERROR
XMTR_RETRY_ON_COMM_STATUS
XMTR_RETRY_ON_DATA
XMTR_RETRY_ON_DEVICE_STATUS
XMTR_RETRY_ON_NO_DEVICE
XMTR_RETRY_ON_RESPONSE_CODE

5.4 Builtins d'action

Les builtins d'action doivent être utilisés uniquement pour les actions appelées par une VARIABLE. Ils permettent d'accéder à la valeur de la variable qui a appelé l'action. Le Tableau 9 donne la liste des builtins d'action.

Tableau 9 – Builtins d'action

Référence	Nom du builtin	Types de données admis pour la variable d'action	Actions admises à partir desquelles le builtin peut être appelé
6.4.1	fgetval	FLOAT, DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

Référence	Nom du built-in	Types de données admis pour la variable d'action	Actions admises à partir desquelles le built-in peut être appelé
6.4.2	fsetval	FLOAT, DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.3	get_date	DATE_AND_TIME, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.4	get_double	DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.5	get_float	FLOAT	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.6	get_signed	INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.7	get_string	ASCII, BITSTRING, BIT_ENUMERATED, ENUMERATED, EUC, INDEX, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.8	get_unsigned	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, UNSIGNED_INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.9	igetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

Référence	Nom du built-in	Types de données admis pour la variable d'action	Actions admises à partir desquelles le built-in peut être appelé
6.4.10	isetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.12	lgetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS
6.4.13	lsetval	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, INTEGER, UNSIGNED_INTEGER, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS
6.4.14	put_date	DATE_AND_TIME, TIME_VALUE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.15	put_double	DOUBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.16	put_float	FLOAT	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.17	put_signed	INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.18	put_string	ASCII, BITSTRING, BIT_ENUMERATED, ENUMERATED, EUC, INDEX, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS

Référence	Nom du builtin	Types de données admis pour la variable d'action	Actions admises à partir desquelles le builtin peut être appelé
6.4.19	put_unsigned	BIT_ENUMERATED, BOOLEAN, ENUMERATED, INDEX, UNSIGNED_INTEGER	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS
6.4.20	sgetval	ASCII, BITSTRING, EUC, PACKED_ASCII, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS
6.4.21	ssetval	ASCII, BITSTRING, EUC, PACKED_ASCII, PASSWORD, VISIBLE	PRE_EDIT_ACTIONS POST_EDIT_ACTIONS PRE_READ_ACTIONS POST_READ_ACTIONS PRE_WRITE_ACTIONS POST_WRITE_ACTIONS REFRESH_ACTIONS POST_RQSTUPDATE_ACTIONS POST_USERCHANGE_ACTIONS

6 Bibliothèque de builtin EDDL

6.1 Généralités

L'Article 6 décrit les sous-programmes builtin, qui peuvent être utilisés dans une EDD. Les builtins prennent en charge les interactions avec l'utilisateur ou le dispositif. Des interactions types avec l'utilisateur sont, par exemple, des entrées de valeurs de VARIABLE par l'utilisateur ou des accusés de réception. Des interactions types avec le dispositif sont, par exemple, l'envoi de commandes au dispositif ou l'interprétation des codes de réponse et des octets de statut. Le présent Article 6 contient la bibliothèque des builtins.

Aucune autre fonction builtin n'est disponible et une EDD conforme doit utiliser uniquement les fonctions de la bibliothèque définies dans la présente spécification. La présente spécification définit les exigences que toutes les applications EDD conformes doivent respecter. Une application EDD conforme doit prendre en charge toutes les fonctions de la bibliothèque indiquées dans la présente spécification qui sont applicables aux profils de communication correspondants. Une application EDD doit prendre en charge toutes les EDD conformes sans exiger de modification de l'EDD conforme et sans informations autres que celles indiquées dans l'EDD conforme.

6.2 Builtins de l'interface utilisateur

6.2.1 Builtin ACKNOWLEDGE

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un message et attendre l'activation d'une touche

Le builtin ACKNOWLEDGE affiche l'invite et attend l'accusé de réception de l'utilisateur.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

ACKNOWLEDGE(prompt)

Les éléments lexicaux du builtin ACKNOWLEDGE sont présentés dans le Tableau 10.

Tableau 10 – Builtin ACKNOWLEDGE

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377.

6.2.2 Builtin acknowledge

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un message et attendre l'activation d'une touche

Le builtin acknowledge affiche l'invite et attend l'accusé de réception de l'utilisateur.

NOTE 1 Les nouvelles descriptions de produits peuvent utiliser le builtin ACKNOWLEDGE, qui prend en charge le référencement direct des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 2 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 3 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

acknowledge(prompt, global_var_ids)

Les éléments lexicaux du builtin acknowledge sont présentés dans le Tableau 11.

Tableau 11 – Builtin acknowledge

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	I	m	Matrice qui contient des identifiants uniques d'instances de VARIABLE.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377.

6.2.3 Builtin DELAY

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un message et un délai

Le builtin DELAY affiche un message d'invite et met en pause pendant le nombre spécifié de secondes. Le délai doit être un nombre positif.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

DELAY(delay_time, prompt)

Les éléments lexicaux du builtin DELAY sont présentés dans le Tableau 12.

Tableau 12 – Builtin DELAY

Nom du paramètre	Type	Direction	Utilisation	Description
delay_time	unsigned integer	I	m	Délai en nombre de secondes.
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	VOID	—	—	—

6.2.4 Builtin delay

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un message et un délai

Le builtin delay affiche un message d'invite et met en pause pendant le nombre spécifié de secondes. Le délai doit être un nombre positif.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Les nouvelles descriptions de produits peuvent utiliser DELAY, qui prend en charge le référencement direct des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

Structure lexicale

delay(delay_time, prompt, global_var_ids)

Les éléments lexicaux du builtin delay sont présentés dans le Tableau 13.

Tableau 13 – Builtin delay

Nom du paramètre	Type	Direction	Utilisation	Description
delay_time	unsigned integer	l	m	Délai en nombre de secondes.
prompt	DD_STRING	l	m	Un message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	l	m	Matrice qui contient des identifiants uniques d'instances de VARIABLE.
<return>	VOID	—	—	—

6.2.5 Builtin delayfor

Profils de communication: FF, ISA100

Objet: Afficher un message et attendre le temps spécifié

Le builtin delayfor imprime une chaîne sur l'écran d'application et attend que la durée allouée expire.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

delayfor(duration, prompt, ids, indices, id_count)

Les éléments lexicaux du builtin delayfor sont présentés dans le Tableau 14.

Tableau 14 – Builtin delayfor

Nom du paramètre	Type	Direction	Utilisation	Description
duration	long	l	m	Délai en nombre de secondes. Il doit s'agir d'un nombre positif.
prompt	DD_STRING	l	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	l	m	ID d'éléments d'une instance de VARIABLE utilisée dans le message d'invite.
indices	unsigned long[]	l	m	ID de membres ou indices de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d'ID de membre ou d'index de matrice.
id_count	long	l	m	Nombre d'éléments de matrice dans les matrices pour les ids et les indices. Défini sur zéro si les matrices pour les ids et les indices sont NULL.

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long ids[2];
unsigned long indices[2];
ids[0] = ids[1] = ITEM_ID(PARAM.OPERATING_TEMP);
indices[0] = MEMBER_ID(HI_LIM);
indices[1] = MEMBER_ID(LO_LIM);
status = delayfor(5, "|en|Operating temperature limits - HI: %{0}
LO: %{1}\n", ids, indices, 2);
```

6.2.6 Builtin delayfor2

Profils de communication: FF, ISA100

Objet: Afficher un message et attendre le temps spécifié

Le builtin delayfor2 imprime une chaîne sur l'écran d'application et attend que la durée allouée expire.

Les valeurs de VARIABLE, les variables de méthode et les libellés pour les VARIABLEs peuvent être incorporés dans la chaîne d'invite avec des informations de formatage. Les paramètres blk_ids, blk_nums, ids, indices et id_count spécifient collectivement les valeurs à incorporer dans l'invite.

Le paramètre prompt peut contenir plusieurs traductions linguistiques comme indiqué ci-dessous.

```
"|en|Is limit correct?|de|Grenze richtig?"
```

Chaque élément de la matrice des blk_ids, block_nums, ids et indices spécifie une valeur à incorporer dans l'invite. Par exemple, dans le cas où trois valeurs sont incorporées dans une invite, les valeurs sont spécifiées comme suit:

```
blk_ids[0], blk_nums[0], ids[0], indices[0]
```

spécifie la première valeur incorporée dans l'invite

```
blk_ids[1], blk_nums [1], ids[1], indices [1]
```

spécifie la deuxième valeur incorporée dans l'invite

```
blk_ids[2], blk_nums [2], ids[2], indices [2]
```

spécifie la troisième valeur incorporée dans l'invite

Certaines des fonctions builtin n'utilisent pas les paramètres blk_ids et blk_nums, tandis que d'autres les utilisent. Lorsque ces paramètres sont utilisés, la fonction builtin extrait les valeurs du bloc spécifié. Lorsqu'ils ne sont pas utilisés, la fonction builtin extrait les valeurs du bloc mis à la portée de la méthode appelante.

Les chaînes d'invite peuvent inclure des spécificateurs de format qui indiquent les valeurs à incorporer dans l'invite. Un spécificateur de format prend l'une des formes suivantes:

```
%format-spezifier{array-index}
```

ou

```
%L{array-index}
```

ou

```
%format-spezifier{local-variable-name}
```

où le spécificateur de format indique un spécificateur de format printf optionnel, tel que défini dans l'ISO/IEC 9899, plus les ajouts présentés dans le Tableau 49. Si aucun spécificateur de format n'est indiqué, l'attribut `DISPLAY_FORMAT` de la valeur est utilisé. Si aucun attribut `DISPLAY_FORMAT` n'est spécifié, un format par défaut est utilisé, en fonction du type de la valeur. `array-index` spécifie un index dans la matrice des `blk_ids`, `blk_nums`, `ids` et `indices`, utilisée pour spécifier la valeur à incorporer dans l'invite. `local-variable-name` spécifie le nom de la variable locale à incorporer dans l'invite. La forme `%L` incorpore le libellé d'une variable, d'un membre d'enregistrement ou d'un élément de matrice dans l'invite, au lieu d'une valeur.

Le paramètre `blk_ids` spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre `blk_nums` spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre `ids` fait référence à une variable, l'élément correspondant du paramètre `indices` est ignoré. Si un élément du paramètre `ids` fait référence à un enregistrement, l'élément correspondant du paramètre `indices` spécifie un membre de l'enregistrement. Si un élément du paramètre `ids` fait référence à une matrice, l'élément correspondant du paramètre `indices` spécifie un élément de la matrice.

Les éléments des paramètres `blk_ids`, `ids` et `indices` sont généralement spécifiés à l'aide des fonctions builtin `ITEM_ID()` et `MEMBER_ID()`. Les éléments du paramètre `blk_nums` sont généralement spécifiés à l'aide des fonctions builtin `get_block_instance_by_object_index()` et `get_block_instance_by_block_tag()`.

Les paramètres `blk_ids`, `blk_nums` et `ids` peuvent être transmis en tant que valeur `NULL` si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur `NULL`, ils doivent tous être transmis en tant que valeur `NULL`.

Le paramètre `indices` peut être transmis en tant que valeur `NULL` si le paramètre `ids` référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

Le paramètre `id_count` doit être transmis en tant que valeur zéro lorsque les paramètres `blk_ids`, `blk_nums` et `ids` sont transmis en tant que valeur `NULL`.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
delayfor2(duration, prompt, blk_ids, blk_nums, ids, indices, id_count)
```

Les éléments lexicaux du builtin `delayfor2` sont présentés dans le Tableau 15.

Tableau 15 – Builtin delayfor2

Nom du paramètre	Type	Direction	Utilisation	Description
duration	long	I	m	Délai en nombre de secondes. Il doit s'agir d'un nombre positif.
prompt	DD_STRING	I	m	Message à afficher. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
blk_ids	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
blk_nums	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES, RECORDs ou ARRAYs utilisés dans l'invite.
indices	unsigned long[]	I	m	ID de membres des RECORDs ou indices de matrice des ARRAYs utilisés dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long blk_ids[2];
unsigned long blk_nums [2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums [0] = blk_nums[1] = 0;
ids[0] = ids[1] = MEMBER_ID(OPERATING_TEMP);
indices[0] = MEMBER_ID(HI_LIM);
indices[1] = MEMBER_ID(LO_LIM);
status = delayfor2(5, "|en|Operating temperature limits - HI: %{0}
LO: %{1}\n", blk_ids, blk_nums, ids, indices, 2);
```

6.2.7 Builtin DISPLAY

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message spécifié à l'écran

Le builtin DISPLAY affiche le message spécifié à l'écran, en mettant à jour en continu les valeurs de variable dynamiques utilisées dans la chaîne. La valeur de variable continue d'être mise à jour jusqu'à ce que l'opérateur effectue l'accusé de réception.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

DISPLAY(prompt)

Les éléments lexicaux du builtin DISPLAY sont présentés dans le Tableau 16.

Tableau 16 – Builtin DISPLAY

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	VOID	—	—	—

6.2.8 Builtin display

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message spécifié à l'écran

Le builtin display affiche le message spécifié à l'écran, en mettant à jour en continu les valeurs de VARIABLE dynamiques utilisées dans la chaîne. La valeur de VARIABLE continue d'être mise à jour jusqu'à ce que l'opérateur effectue l'accusé de réception.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Les nouvelles descriptions de produits peuvent utiliser DISPLAY, qui prend en charge le référencement direct des variables de méthode et des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

Structure lexicale

display(prompt, global_var_ids)

Les éléments lexicaux du builtin display sont présentés dans le Tableau 17.

Tableau 17 – Builtin display

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	I	m	Matrice qui contient des identifiants uniques d'instances de VARIABLE.
<return>	VOID	—	—	—

6.2.9 Builtin display_bitenum

Profils de communication: CS, GPE, PB, PN

Objet: Afficher la chaîne d'énumération

Le builtin display_bitenum affiche une représentation textuelle d'une VARIABLE BIT_ENUMERATED et attend l'accusé de réception de l'utilisateur.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_bitenum(bitenum_variable)
```

Les éléments lexicaux du builtin display_bitenum sont présentés dans le Tableau 18.

Tableau 18 – Builtin display_bitenum

Nom du paramètre	Type	Direction	Utilisation	Description
bitenum_variable	Reference	I	m	VARIABLE BIT_ENUMERATED qui doit être affichée.
<return>	VOID	—	—	—

6.2.10 Builtin display_builtin_error

Profils de communication: FF, ISA100

Objet: Afficher la description de l'erreur builtin et attendre l'accusé de réception

Le builtin display_builtin_error affiche sur le dispositif d'affichage de l'application le texte associé à une erreur builtin spécifiée. Le builtin display ne retourne pas le contrôle à la méthode tant que l'utilisateur n'accuse pas réception du message.

Les codes d'erreur et leurs descriptions textuelles sont spécifiés dans le Tableau 376.

Ce builtin constitue un moyen de débogage des appels de programmes builtin qui retournent une erreur. Il convient de ne jamais l'utiliser dans une méthode de production finale, car il convient que les types des erreurs retournées par les builtins ne soient jamais reçus dans une méthode totalement déboguée.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_builtin_error(error)
```

Les éléments lexicaux du builtin display_builtin_error sont présentés dans le Tableau 19.

Tableau 19 – Builtin display_builtin_error

Nom du paramètre	Type	Direction	Utilisation	Description
error	long	I	m	Code pour une erreur builtin.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
if (status != BLTIN_SUCCESS)
{
    display_builtin_error(status);
}
```

6.2.11 Builtin display_comm_error

Profils de communication: FF, ISA100

Objet: Afficher la description de l'erreur de communication et attendre l'accusé de réception

Le builtin display_comm_error affiche la chaîne de texte associée à une erreur de communication sur le dispositif d'affichage de l'application.

NOTE 1 Une erreur de communication égale à zéro n'est pas assimilée à une erreur de communication.

NOTE 2 Le builtin ne retourne pas le contrôle à la méthode tant que l'utilisateur n'accuse pas réception du message.

En interne, ce builtin appelle le builtin get_comm_error_string pour obtenir la chaîne de texte associée et le builtin get_acknowledgement pour afficher la chaîne de texte associée et attendre l'accusé de réception de l'utilisateur.

Les codes d'erreur de communication sont définis par les consortiums. Leurs chaînes associées peuvent être définies dans un dictionnaire de texte.

NOTE 3 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 4 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_comm_error(error)
```

Les éléments lexicaux du builtin display_comm_error sont présentés dans le Tableau 20.

Tableau 20 – Builtin display_comm_error

Nom du paramètre	Type	Direction	Utilisation	Description
error	unsigned long	I	m	Code pour une erreur de communication.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
if (status == BLTIN_FAIL_COMM_ERROR)
{
    status = display_comm_error(get_comm_error());
}
```

L'exemple utilise le builtin get_comm_error() pour obtenir le code d'erreur de communication utilisé comme argument de ce builtin.

6.2.12 Builtin display_comm_status

Profils de communication: HART

Objet: Afficher la chaîne de statut de communication

Le builtin display_comm_status affiche la ou les chaînes de description de la VARIABLE de statut de communication désignée par le paramètre comm_status_value. La chaîne affichée reste à l'écran jusqu'à ce que l'opérateur effectue l'accusé de réception.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_comm_status(comm_status_value)`

Les éléments lexicaux du builtin `display_comm_status` sont présentés dans le Tableau 21.

Tableau 21 – Builtin `display_comm_status`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>comm_status_value</code>	int	l	m	Code pour une erreur de communication.
<return>	VOID	—	—	—

6.2.13 Builtin `display_device_status`

Profils de communication: HART

Objet: Afficher la chaîne de statut du dispositif

Le builtin `display_device_status` affiche la ou les chaînes de description de la VARIABLE de statut du dispositif désignée par le paramètre `device_status_value`. La chaîne affichée reste à l'écran jusqu'à ce que l'opérateur effectue l'accusé de réception.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_device_status(device_status_value)`

Les éléments lexicaux du builtin `display_device_status` sont présentés dans le Tableau 22.

Tableau 22 – Builtin `display_device_status`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>device_status_value</code>	int	l	m	Code pour le statut du dispositif.
<return>	VOID	—	—	—

6.2.14 Builtin `display_dynamics`

Profils de communication: FF, ISA100

Objet: Afficher et mettre à jour les valeurs de VARIABLE

Le builtin `display_dynamics` affiche un message en continu sur le dispositif de sortie de l'application. Le builtin met à jour les valeurs spécifiées dans le message jusqu'à ce qu'il soit interrompu par l'utilisateur. Le contrôle n'est pas retourné à la méthode tant qu'il n'est pas interrompu par l'utilisateur.

Le principal objectif de l'appel est d'afficher des informations de valeur du dispositif que l'utilisateur souhaite examiner pendant une période.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_dynamics(prompt, ids, indices, id_count)`

Les éléments lexicaux du builtin `display_dynamics` sont présentés dans le Tableau 23.

Tableau 23 – Builtin `display_dynamics`

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d'ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long ids[2];
unsigned long indices[2];
ids[0] = ITEM_ID(PARAM.OUT);
ids[1] = ITEM_ID(PARAM.BKCAL_IN);
indices[0] = indices[1] = MEMBER_ID(VALUE);
status = display_dynamics("|en|PID.OUT: %{0} PID.BKCAL_IN: %{1}, ids,
indices, 2);
```

6.2.15 Builtin `display_dynamics2`

Profils de communication: FF, ISA100

Objet: Afficher et mettre à jour les valeurs de VARIABLE

Le builtin `display_dynamics2` affiche un message en continu sur le dispositif de sortie de l'application. Le builtin met à jour les valeurs spécifiées dans le message jusqu'à ce qu'il soit interrompu par l'utilisateur. Le contrôle n'est pas retourné à la méthode tant qu'il n'est pas interrompu par l'utilisateur. Le principal objectif de l'appel est d'afficher des informations de valeur du dispositif que l'utilisateur souhaite examiner pendant une période.

Les paramètres `blk_ids`, `blk_nums`, `ids`, `indices` et `id_count` spécifient collectivement les valeurs à incorporer dans l'invite. Le paramètre `blk_ids` spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre `blk_nums` spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre `ids` fait référence à une variable, l'élément correspondant du paramètre `indices` est ignoré. Si un élément du paramètre `ids` fait référence à un enregistrement, l'élément correspondant du paramètre `indices` spécifie un membre de l'enregistrement. Si un élément du paramètre `ids` fait référence à une matrice, l'élément correspondant du paramètre `indices` spécifie un élément de la matrice.

Les éléments des paramètres `blk_ids`, `ids` et `indices` sont généralement spécifiés à l'aide des fonctions builtin `ITEM_ID()` et `MEMBER_ID()`. Les éléments du paramètre `blk_nums` sont généralement spécifiés à l'aide des fonctions builtin `get_block_instance_by_object_index()` et `get_block_instance_by_block_tag()`.

Les paramètres `blk_ids`, `blk_nums` et `ids` peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_dynamics2(prompt, blk_ids, blk_nums, ids, indices, id_count)`

Les éléments lexicaux du builtin `display_dynamics2` sont présentés dans le Tableau 24.

Tableau 24 – Builtin `display_dynamics2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>prompt</code>	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<code>blk_ids</code>	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
<code>blk_nums</code>	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
<code>ids</code>	unsigned long[]	I	m	ID d'éléments des VARIABLES, RECORDs ou ARRAYs utilisés dans l'invite.
<code>indices</code>	unsigned long[]	I	m	ID de membres des RECORDs ou indices de matrice des ARRAYs utilisés dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
<code>id_count</code>	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
unsigned long blk_ids[2];
unsigned long blk_nums [2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] =
ITEM_ID(__proportional_integral_derivative_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(OUT);
ids[1] = MEMBER_ID(BKCAL_IN);
indices[0] = indices[1] = MEMBER_ID(VALUE);
status = display_dynamics2("|en|PID.OUT: %{0} PID.BKCAL_IN: %{1}, blk_ids,
blk_nums, ids, indices, 2);

```

6.2.16 Builtin display_message**Profils de communication: FF, ISA100****Objet: Afficher un message qui n'exige pas d'accusé de réception de l'utilisateur**

Le builtin display_message affiche un message sur le dispositif de sortie de l'application. Il retourne le contrôle à la méthode sans exiger d'accusé de réception de l'utilisateur.

Cet appel est utilisé pour informer l'utilisateur de changements de statut du système qui n'exigent aucune action. Il convient que le message reste disponible pour examen au moins jusqu'à ce qu'un appel ultérieur remplace potentiellement le message par un nouveau message ou que la méthode se termine.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_message(prompt, ids, indices, id_count)
```

Les éléments lexicaux du builtin display_message sont présentés dans le Tableau 25.

Tableau 25 – Builtin display_message

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long ids[1];
unsigned long indices[1];
status = display_message("|en|All clear\n", ids, indices, 0);
```

6.2.17 Builtin display_message2**Profils de communication: FF, ISA100****Objet: Afficher un message qui n'exige pas d'accusé de réception de l'utilisateur**

Le builtin `display_message2` affiche un message sur le dispositif de sortie de l'application. Il retourne le contrôle à la méthode sans exiger d'accusé de réception de l'utilisateur. Cet appel est utilisé pour informer l'utilisateur de changements de statut du système qui n'exigent aucune action. Il convient que le message reste disponible pour examen au moins jusqu'à ce qu'un appel ultérieur remplace potentiellement le message par un nouveau message ou que la méthode se termine.

Le paramètre `blk_ids` spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre `blockInstances` spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre `ids` fait référence à une variable, l'élément correspondant du paramètre `indices` est ignoré. Si un élément du paramètre `ids` fait référence à un enregistrement, l'élément correspondant du paramètre `indices` spécifie un membre de l'enregistrement. Si un élément du paramètre `ids` fait référence à une matrice, l'élément correspondant du paramètre `indices` spécifie un élément de la matrice.

Les éléments des paramètres `blk_ids`, `ids` et `indices` sont généralement spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Les éléments du paramètre `blk_nums` sont généralement spécifiés à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Les paramètres `blk_ids`, `blk_nums` et `ids` peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

Le paramètre `indices` peut être transmis en tant que valeur NULL si le paramètre `ids` référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

Le paramètre `count` doit être transmis en tant que valeur zéro lorsque les paramètres `blk_ids`, `blk_nums` et `indices` sont transmis en tant que valeur NULL.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_message2(prompt, blk_ids, blk_nums, ids, indices, id_count)
```

Les éléments lexicaux du builtin `display_message2` sont présentés dans le Tableau 26.

Tableau 26 – Builtin display_message2

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
blk_ids	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
blk_nums	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES, RECORDs ou ARRAYs utilisés dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d'ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
unsigned long blk_ids[1];
unsigned long blk_nums[1];
unsigned long ids[1];
unsigned long indices[1];
blk_ids[0] = ITEM_ID(__analog_input_block);
blk_nums[0] = 0;
ids[0] = MEMBER_ID(MODE_BLK);
indices[0] = MEMBER_ID(TARGET);
status = display_message2("|en|Target Mode is %{0}\n", blk_ids, blk_nums,
ids, indices, 1);

```

6.2.18 Builtin display_response_code**Profils de communication: FF, ISA100****Objet: Afficher la description d'un code de réponse et attendre un accusé de réception**

Le builtin display_response_code affiche la description d'un code de réponse pour un élément. Le builtin display ne retourne pas le contrôle à la méthode tant que l'utilisateur n'accuse pas réception du message.

En interne, ce builtin appelle le builtin get_response_code_string pour obtenir la chaîne de texte associée et le builtin get_acknowledgement pour afficher la chaîne de texte associée et attendre l'accusé de réception de l'utilisateur.

Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application. Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

L'ID de paramètre et le member_id désignent un paramètre auquel peut être associé un code de réponse après sa dernière lecture ou écriture. L'ID de paramètre et le member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la variable en question.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
display_response_code(id, member_id, code)
```

Les éléments lexicaux du builtin display_response_code sont présentés dans le Tableau 27.

Tableau 27 – Builtin display_response_code

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'élément de l'élément qui a généré le code de réponse.
member_id	unsigned long	I	m	ID de membre de l'élément qui a généré le code de réponse.
code	unsigned long	I	m	Code de réponse reçu du builtin get_response_code.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long id;
unsigned long member;
unsigned long code;
if (status == BLTIN_FAIL_RESPONSE_CODE)
{
    get_response_code(&code, &id, &member);
    status = display_response_code(id, member, code);
}
```

L'exemple utilise le builtin get_response_code() pour obtenir le code de réponse retourné lors de la précédente demande de communication.

6.2.19 Builtin display_response_status

Profils de communication: HART

Objet: Afficher la chaîne de statut de réponse

Le builtin display_response_status affiche la chaîne associée à la valeur spécifiée de l'octet response_code reçu du COMMAND NUMBER spécifié. La chaîne affichée reste à l'écran jusqu'à ce que l'opérateur effectue l'accusé de réception.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_response_status(command_num, response_code_value)`

Les éléments lexicaux du builtin `display_response_status` sont présentés dans le Tableau 28.

Tableau 28 – Builtin `display_response_status`

Nom du paramètre	Type	Direction	Utilisation	Description
Command_name	Int	I	m	NUMBER de la COMMAND ayant envoyé le statut de réponse.
response_code_value	int	I	m	Code de réponse à afficher dans une invite.
<return>	VOID	—	—	—

6.2.20 Builtin `display_xmtr_status`

Profils de communication: HART

Objet: Afficher la chaîne de statut du transmetteur

Le builtin `display_xmtr_status` affiche une chaîne de description associée au bit indiqué par le paramètre `status_mask` dans la VARIABLE désignée par le paramètre `bit_enum_var_name`. La chaîne s'affiche, que le bit indiqué par le paramètre `status_mask` soit défini ou non. La chaîne affichée reste à l'écran jusqu'à ce que l'opérateur effectue l'accusé de réception.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`display_xmtr_status(bit_enum_var_name, status_mask)`

Les éléments lexicaux du builtin `display_xmtr_status` sont présentés dans le Tableau 29.

Tableau 29 – Builtin `display_xmtr_status`

Nom du paramètre	Type	Direction	Utilisation	Description
bit_enum_var_name	int	I	m	Référence à une instance de VARIABLE de type BIT_ENUMERATED.
status_mask	int	I	m	Masque de bits qui indique le bit à afficher.
<return>	VOID	—	—	—

6.2.21 Builtin `edit_device_value`

Profils de communication: FF, ISA100

Objet: Afficher une valeur de dispositif définie dans EDDL et laisser l'utilisateur la modifier

Le builtin `edit_device_value` affiche une invite avec une valeur du dispositif spécifiée par le param `id` et le param `member_id`. L'utilisateur peut éditer la valeur affichée.

Une valeur du dispositif peut résider dans la mémoire cache d'une application EDD en mémoire cache, et la valeur ne doit pas être écrite dans le dispositif tant qu'une méthode n'est pas terminée ou que la valeur n'est pas explicitement envoyée au dispositif par la méthode.

L'application édite et vérifie les valeurs entrées pour la valeur du dispositif. Cette édition est fondée sur des types arithmétiques de VARIABLE définis dans l'EDD d'origine pour chaque VARIABLE. Les attributs EDDL associés aux VARIABLEs peuvent comprendre:

- HANDLING
- DISPLAY_FORMAT
- EDIT_FORMAT
- MIN_VALUE
- MAX_VALUE

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

edit_device_value(prompt, ids, indices, id_count, param_id, param_member_id)

Les éléments lexicaux du builtin edit_device_value sont présentés dans le Tableau 30.

Tableau 30 – Builtin edit_device_value

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLEs utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLEs utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
param_id	unsigned long	I	m	ID d'élément de la VARIABLE à modifier.
param_member_id	unsigned long	I	m	ID de membre de la VARIABLE à modifier.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long id;
unsigned long ids[1];
unsigned long indices[1];
id = ITEM_ID(PARAM.HI_LIM);
status = edit_device_value("|en|Enter new HI_LIM value:\n", ids, indices,
0, id, 0);
status = send_value(id, 0);
```

6.2.22 Builtin edit_device_value2

Profils de communication: FF, ISA100

Objet: Afficher une valeur de dispositif définie dans EDDL et laisser l'utilisateur la modifier

Le builtin edit_device_value2 affiche une invite et permet à l'utilisateur de modifier la variable, le membre d'enregistrement ou l'élément de matrice spécifié par les paramètres blk_id, blk_num, param_id et param_member_id.

Une valeur du dispositif peut résider dans la mémoire cache d'une application EDD en mémoire cache, et la valeur ne doit pas être écrite dans le dispositif tant qu'une méthode n'est pas terminée ou que la valeur n'est pas explicitement envoyée au dispositif par la méthode.

Les paramètres blk_ids, blk_nums, ids, indices et count spécifient collectivement les valeurs à incorporer dans l'invite.

L'application édite et vérifie les valeurs entrées pour la valeur du dispositif. Cette édition est fondée sur des attributs de variable, de membre d'enregistrement ou d'élément de matrice définis dans l'EDD d'origine pour chaque VARIABLE. Les attributs EDDL associés aux VARIABLES peuvent comprendre:

- HANDLING
- DISPLAY_FORMAT
- EDIT_FORMAT
- MIN_VALUE
- MAX_VALUE

Le paramètre blk_ids spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre blk_nums spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre ids fait référence à une variable, l'élément correspondant du paramètre indices est ignoré. Si un élément du paramètre ids fait référence à un enregistrement, l'élément correspondant du paramètre indices spécifie un membre de l'enregistrement. Si un élément du paramètre ids fait référence à une matrice, l'élément correspondant du paramètre indices spécifie un élément de la matrice.

Les éléments des paramètres blk_ids, ids et indices sont généralement spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Les éléments du paramètre blk_nums sont généralement spécifiés à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

Les paramètres blk_ids, blk_nums et ids peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

Le paramètre indices peut être transmis en tant que valeur NULL si le paramètre ids référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

Le paramètre count doit être transmis en tant que valeur zéro lorsque les paramètres blk_ids, blk_nums et ids sont transmis en tant que valeur NULL.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

`edit_device_value2(prompt, blk_ids, blk_nums, ids, indices, id_count, blk_id, blk_num, param_id, param_member_id)`

Les éléments lexicaux du builtin `edit_device_value2` sont présentés dans le Tableau 31.

Tableau 31 – Builtin `edit_device_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>prompt</code>	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<code>blk_ids</code>	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
<code>blk_nums</code>	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
<code>ids</code>	unsigned long[]	I	m	ID d'éléments des VARIABLES, RECORDs ou ARRAYs utilisés dans le message d'invite.
<code>indices</code>	unsigned long[]	I	m	ID de membres ou index de matrice des membres d'enregistrement et/ou des éléments de matrice utilisés dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
<code>id_count</code>	long	I	m	Nombre d'éléments dans les matrices <code>blk_ids</code> , <code>blk_nums</code> , <code>ids</code> et <code>indices</code> . Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance de BLOCK_A à modifier.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à modifier.
<code>param_id</code>	unsigned long	I	m	ID d'élément de la VARIABLE à modifier.
<code>param_member_id</code>	unsigned long	I	m	ID de membre de la VARIABLE à modifier.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long blk_ids[1];
unsigned long blk_nums[1];
unsigned long ids[1];
unsigned long indices[1];
unsigned long blk_id;
unsigned long blk_num;
unsigned long param_id;
blk_id = ITEM_ID(__analog_input_block);
blk_num = 0;
param_id = MEMBER_ID(HI_LIM);
status = edit_device_value2("|en|Enter new HI_LIM value:\n", blk_ids,
blk_nums, ids, indices, 0, blk_id, blk_num, param_id, 0);
```

6.2.23 Builtin edit_local_value

Profils de communication: FF, ISA100

Objet: Afficher une variable de méthode et laisser l'utilisateur la modifier

Le builtin edit_local_value affiche une invite avec le nom de la variable de méthode spécifié par le paramètre local_var. Le paramètre local_var est une chaîne de caractères C à terminaison null. L'utilisateur peut éditer la valeur affichée de la variable de méthode.

L'application EDD est responsable de la prévention du retour de valeurs non valides en effectuant des vérifications fondées sur le type de variable de méthode uniquement. La méthode doit effectuer une vérification d'intervalle sur les valeurs retournées.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

edit_local_value(prompt, ids, indices, id_count, local_var)

Les éléments lexicaux du builtin edit_local_value sont présentés dans le Tableau 32.

Tableau 32 – Builtin edit_local_value

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
local_var	char[]	I	m	Nom de la VARIABLE locale qui peut être modifiée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long ids[1];
unsigned long indices[1];
float offset;
offset = 1.0;
status = edit_local_value("|en|Please enter offset...\n", ids, indices, 0,
"offset");
```

6.2.24 Builtin edit_local_value2

Profils de communication: FF, ISA100

Objet: Afficher une variable de méthode et laisser l'utilisateur la modifier

Le builtin edit_local_value2 affiche une invite avec le nom de la variable de méthode spécifié par le paramètre local_var. Le paramètre local_var est une chaîne de caractères C à terminaison null. L'utilisateur peut éditer la valeur affichée de la variable de méthode.

Les paramètres blk_ids, blk_nums, ids, indices et count spécifient collectivement les valeurs à incorporer dans l'invite.

L'application EDD est responsable de la prévention du retour de valeurs non valides en effectuant des vérifications fondées sur le type de variable de méthode uniquement. La méthode doit effectuer une vérification d'intervalle sur les valeurs retournées.

Le paramètre blk_ids spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre blk_nums spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre ids fait référence à une variable, l'élément correspondant du paramètre indices est ignoré. Si un élément du paramètre ids fait référence à un enregistrement, l'élément correspondant du paramètre indices spécifie un membre de l'enregistrement. Si un élément du paramètre ids fait référence à une matrice, l'élément correspondant du paramètre indices spécifie un élément de la matrice.

Les éléments des paramètres blk_ids, ids et indices sont généralement spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Les éléments du paramètre blk_nums sont généralement spécifiés à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

Les paramètres blk_ids, blk_nums et ids peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

Le paramètre indices peut être transmis en tant que valeur NULL si le paramètre ids référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

Le paramètre count doit être transmis en tant que valeur zéro lorsque les paramètres blk_ids, blk_nums et ids sont transmis en tant que valeur NULL.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
edit_local_value2(prompt, blk_ids, blk_nums, ids, indices, id_count,
local_var)
```

Les éléments lexicaux du builtin edit_local_value2 sont présentés dans le Tableau 33.

Tableau 33 – Builtin edit_local_value2

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
blk_ids	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
blk_nums	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou index de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'index de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
local_var	char[]	I	m	Nom de la VARIABLE locale qui peut être modifiée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
float offset;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[0];
offset = 1.0;
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(LOW_LIMIT);
ids[1] = MEMBER_ID(HIGH_LIMIT);
indices[0] = indices[1] = 0;
status = edit_local_value2("|en|Please enter offset between %{0}
and %{1}\n", blk_ids, blk_nums, ids, indices, 2, "offset");

```

6.2.25 Builtin get_acknowledgement**Profils de communication: FF, ISA100****Objet: Afficher un message qui exige un accusé de réception de l'utilisateur**

Le builtin get_acknowledgement affiche le message d'invite fourni sur le dispositif d'affichage de l'application et retourne à la méthode lorsque l'utilisateur a accusé réception du message.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
get_acknowledgement(prompt, ids, indices, id_count)
```

Les éléments lexicaux du builtin `get_acknowledgement` sont présentés dans le Tableau 34.

Tableau 34 – Builtin `get_acknowledgement`

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou indices de matrice des VARIABLES utilisées dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'indice de matrice.
id_count	long	I	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long ids[1];
unsigned long indices[1];
status = get_acknowledgement("|en|W A R N\n loop should be removed from
auto mode\n", ids, indices, 0,);
```

6.2.26 Builtin `get_acknowledgement2`

Profils de communication: FF, ISA100

Objet: Afficher un message qui exige un accusé de réception de l'utilisateur

Le builtin `get_acknowledgement2` affiche le message d'invite fourni sur le dispositif d'affichage de l'application et retourne à la méthode lorsque l'utilisateur a accusé réception du message. Les paramètres `blk_ids`, `blk_nums`, `ids`, `indices` et `count` spécifient collectivement les valeurs à incorporer dans l'invite.

Le paramètre `blk_ids` spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre `blk_nums` spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre `ids` fait référence à une variable, l'élément correspondant du paramètre `indices` est ignoré. Si un élément du paramètre `ids` fait référence à un enregistrement, l'élément correspondant du paramètre `indices` spécifie un membre de l'enregistrement. Si un élément du paramètre `ids` fait référence à une matrice, l'élément correspondant du paramètre `indices` spécifie un élément de la matrice.

Les éléments des paramètres `blk_ids`, `ids` et `indices` sont généralement spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Les éléments du paramètre `blk_nums` sont généralement spécifiés à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Les paramètres `blk_ids`, `blk_nums` et `ids` peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

Le paramètre indices peut être transmis en tant que valeur NULL si le paramètre ids référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
get_acknowledgement2(prompt, blk_ids, blk_nums, ids, indices, id_count)
```

Les éléments lexicaux du builtin get_acknowledgement2 sont présentés dans le Tableau 35.

Tableau 35 – Builtin get_acknowledgement2

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	l	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
blk_ids	unsigned long[]	l	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
blk_nums	unsigned long[]	l	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
ids	unsigned long[]	l	m	ID d'éléments des VARIABLES, RECORD ou ARRAYs utilisés dans le message d'invite.
indices	unsigned long[]	l	m	ID de membres ou indices de matrice des membres et/ou des éléments de matrice utilisés dans le message d'invite. Défini sur zéro s'il n'y a pas d>ID de membre ou d'indice de matrice.
id_count	long	l	m	Nombre d'éléments de matrice dans les matrices pour les ID et les indices. Défini sur zéro si les matrices pour les ID et les indices sont NULL.
<return>	long	o	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[2];
blk_ids[0] = blk_ids[1] = ITEM_ID(transducer_block);
blk_nums[0] = blk_nums[1] = 0;
ids[0] = MEMBER_ID(LOW_LIMIT);
ids[1] = MEMBER_ID(HIGH_LIMIT);
indices[0] = indices[1] = 0;
status = get_acknowledgement2("Setting LOW = %{0} and HIGH = %{1}\n",
blk_ids, blk_nums, ids, indices, 2);
```


6.2.27 Builtin GET_DEV_VAR_VALUE

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message d'invite spécifié et permettre à l'utilisateur d'éditer la valeur d'une VARIABLE

Le builtin GET_DEV_VAR_VALUE affiche le message d'invite spécifié et permet à l'utilisateur d'éditer la valeur d'une VARIABLE. La copie éditée de la valeur de la VARIABLE est mise à jour lorsque la nouvelle valeur est entrée, mais n'est pas envoyée au dispositif. Cela doit être effectué de manière explicite en utilisant une des fonctions de commande d'envoi.

PRE_EDIT_ACTIONS de la VARIABLE est appelé avant invite de la nouvelle valeur. POST_EDIT_ACTIONS est appelé après l'entrée d'une nouvelle valeur.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

GET_DEV_VAR_VALUE(prompt, device_var_name)

Les éléments lexicaux du builtin GET_DEV_VAR_VALUE sont présentés dans le Tableau 36.

Tableau 36 – Builtin GET_DEV_VAR_VALUE

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
device_var_name	reference	I	m	Nom référencé de la VARIABLE, qui peut être édité.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la modification de la variable a abouti ou BI_ERROR si une erreur s'est produite lors de l'entrée de la nouvelle valeur ou de l'accès à la variable spécifiée.

6.2.28 Builtin get_dev_var_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message d'invite spécifié et permettre à l'utilisateur d'éditer⁵ la valeur d'une VARIABLE

Le builtin get_dev_var_value affiche le message d'invite spécifié et permet à l'utilisateur d'éditer la valeur d'une VARIABLE. La copie éditée de la valeur de la VARIABLE est mise à jour lorsque la nouvelle valeur est entrée, mais n'est pas envoyée au dispositif. Cela doit être effectué de manière explicite en utilisant une des fonctions de commande d'envoi.

⁵ L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

PRE_EDIT_ACTIONS de la VARIABLE est appelé avant invite de la nouvelle valeur. POST_EDIT_ACTIONS est appelé après l'entrée d'une nouvelle valeur.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Les nouvelles descriptions de produits peuvent utiliser GET_DEV_VAR_VALUE, qui prend en charge le référencement direct des variables de méthode et des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

Structure lexicale

get_dev_var_value(prompt, global_var_ids, device_var_name)

Les éléments lexicaux du builtin get_dev_var_value sont présentés dans le Tableau 37.

Tableau 37 – Builtin get_dev_var_value

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	I	m	Matrice qui contient des identifiants uniques d'instance de VARIABLE.
device_var_name	reference	I	m	Nom de la VARIABLE, qui peut être édité.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la modification de la variable a abouti ou BI_ERROR si une erreur s'est produite lors de l'entrée de la nouvelle valeur ou de l'accès à la variable spécifiée.

6.2.29 Builtin GET_LOCAL_VAR_VALUE

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message d'invite spécifié et permettre à l'utilisateur d'éditer la valeur d'une variable de méthode

Le builtin GET_LOCAL_VAR_VALUE affiche le message d'invite spécifié et permet à l'utilisateur d'éditer la valeur d'une variable de méthode.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

GET_LOCAL_VAR_VALUE(prompt, local_var_name)

Les éléments lexicaux du builtin GET_LOCAL_VAR_VALUE sont présentés dans le Tableau 38.

Tableau 38 – Builtin GET_LOCAL_VAR_VALUE

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
local_var_name	referenced	I	m	Nom de la variable de méthode à éditer.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la modification de la variable a abouti et BI_ERROR si une erreur s'est produite lors de l'entrée de la nouvelle valeur ou de l'accès à la variable spécifiée.

6.2.30 Builtin get_local_var_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message d'invite spécifié et permettre à l'utilisateur d'éditer la valeur d'une variable de méthode

Le builtin get_local_var_value affiche le message d'invite spécifié et permet à l'utilisateur d'éditer la valeur d'une variable de méthode.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Les nouvelles descriptions de produits peuvent utiliser GET_LOCAL_VAR_VALUE, qui prend en charge le référencement direct des variables de méthode et des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

Structure lexicale

get_local_var_value(prompt, global_var_ids, local_var_name)

Les éléments lexicaux du builtin get_local_var_value sont présentés dans le Tableau 39.

Tableau 39 – Builtin get_local_var_value

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	I	o	Matrice qui contient des identifiants uniques d'instance de VARIABLE.
local_var_name	referenced	I	o	Nom de la VARIABLE locale à éditer.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la modification de la variable a abouti et BI_ERROR si une erreur s'est produite lors de l'entrée de la nouvelle valeur ou de l'accès à la variable spécifiée.

6.2.31 Builtin Menu

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un menu à l'utilisateur

Le builtin Menu affiche un MENU et attend que l'utilisateur en accuse réception à l'aide d'une seule option (par exemple, "OK").

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Profils de communication: HART

Le MENU référencé depuis ce builtin ne doit pas contenir d'autres méthodes ni d'édérations d'affichages en tant qu'éléments de menus.

Le MENU doit être traité comme un MENU avec STYLE DIALOG quel que soit le STYLE réel du MENU et tous les éléments de sous-menu de type MENU doivent être de style GROUP.

Profils de communication: CS, GPE, PB, PN

Le MENU doit être traité comme un MENU avec STYLE DIALOG quel que soit le STYLE réel du MENU.

Structure lexicale

Menu (menu)

Si l'application EDD est incapable d'afficher le menu, une erreur est retournée.

Les éléments lexicaux du builtin Menu sont présentés dans le Tableau 40.

Tableau 40 – Builtin Menu

Nom du paramètre	Type	Direction	Utilisation	Description
menu	reference	I	m	MENU à afficher.
<return>	long	O	m	Profils de communication: CS, GPE, HART, PB, PN Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la fonction a abouti et BI_ERROR si elle a échoué.

6.2.32 Builtin MenuDisplay (version A)

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher un menu à l'utilisateur et fournir une liste d'options à choisir par l'utilisateur

Le builtin MenuDisplay affiche un MENU et attend un accusé de réception de l'utilisateur par le choix d'une option dans la liste d'options fournie. Par ailleurs, pour éviter toute confusion de l'opérateur, il convient qu'option_list n'utilise pas d'expressions telles que "Abort" et "Cancel". Lorsqu'un choix est effectué, la position de l'option dans la liste d'options est retournée. La position dans la liste est à base zéro. Par exemple, si la liste fournie contient deux éléments et que le premier élément est choisi, cette fonction retourne un 0. Si le deuxième élément est choisi, un 1 est retourné.

Les options sont transmises à la fonction sous la forme d'une chaîne unique délimitée par des points-virgules entre les options. Si aucune option n'est proposée, la fonction retourne une erreur.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction. Le mécanisme d'annulation permettant d'interrompre la méthode doit être fourni par l'application EDD.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Certaines applications EDD à affichage limité ne sont capables d'afficher que 4 boutons. Par conséquent, lorsque trois boutons définis par un développeur sont inclus (plus le mécanisme d'interruption normalisé), il est possible que le bouton d'aide (Help) ne soit plus disponible.

Profils de communication: HART

Trois options définies par l'utilisateur au plus sont admises, à l'exception du mécanisme d'annulation qui doit être fourni par l'application EDD.

Le MENU doit être traité comme un MENU avec STYLE DIALOG quel que soit le STYLE réel du MENU et tous les éléments de sous-menu de type MENU doivent être de style GROUP. Le MENU référencé depuis ce builtin ne doit pas contenir d'autres méthodes ni d'édérations d'affichages en tant qu'éléments de menus.

Profils de communication: CS, GPE, PB, PN

Le MENU doit être traité comme un MENU avec STYLE DIALOG quel que soit le STYLE réel du MENU.

Structure lexicale

MenuDisplay(menu, options, selection)

Si l'application EDD est incapable d'afficher le menu, une erreur est retournée.

Les éléments lexicaux du builtin MenuDisplay sont présentés dans le Tableau 41.

Tableau 41 – Builtin MenuDisplay

Nom du paramètre	Type	Direction	Utilisation	Description
menu	reference	I	m	MENU à afficher.
options	DD_STRING	I	m	Chaîne d'options, constituée de texte délimité, qui définit un ensemble d'options présentées à l'utilisateur. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
selection	long*	O	m	Option base zéro choisie (voir les paramètres d'option).
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si la fonction a abouti et BI_ERROR si elle a échoué. Si BI_ERROR est retourné, la valeur de sélection n'est pas définie.

6.2.33 Builtin MenuDisplay (version B)

Profils de communication: FF, ISA100

Objet: Afficher un menu à l'utilisateur et fournir une liste d'options à choisir par l'utilisateur

Le builtin MenuDisplay affiche un MENU et attend un accusé de réception de l'utilisateur par le choix d'une option dans la liste d'options fournie. Par ailleurs, pour éviter toute confusion de l'opérateur, il convient qu'option_list n'utilise pas d'expressions telles que "Abort" et "Cancel". Lorsqu'un choix est effectué, la position de l'option dans la liste d'options est retournée. La position dans la liste est à base zéro. Par exemple, si la liste fournie contient deux éléments et que le premier élément est choisi, cette fonction retourne un 0. Si le deuxième élément est choisi, un 1 est retourné.

Les options sont transmises à la fonction sous la forme d'une chaîne unique délimitée par des points-virgules entre les options. Si aucune option n'est proposée, la fonction retourne une erreur.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Certaines applications EDD à affichage limité ne sont capables d'afficher que 4 boutons. Par conséquent, lorsque trois boutons définis par un développeur sont inclus (plus le mécanisme d'interruption normalisé), il est possible que le bouton d'aide (Help) ne soit plus disponible.

Trois options définies par l'utilisateur au plus sont admises, à l'exception du mécanisme d'annulation qui doit être fourni par l'application EDD.

Le MENU doit être traité comme un MENU avec STYLE DIALOG quel que soit le STYLE réel du MENU. Le MENU référencé depuis ce builtin ne doit pas contenir d'autres méthodes ni d'édérations d'affichages en tant qu'éléments de menus.

Structure lexicale

```
MenuDisplay(menu, options, selection)
```

Si l'application EDD est incapable d'afficher le menu, une erreur est retournée.

Les éléments lexicaux du builtin MenuDisplay sont présentés dans le Tableau 42.

Tableau 42 – Builtin MenuDisplay

Nom du paramètre	Type	Direction	Utilisation	Description
menu	unsigned long	I	m	ID d'élément du MENU à afficher.
options	DD_STRING	I	m	Chaîne d'options, constituée de texte délimité, qui définit un ensemble d'options présentées à l'utilisateur. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
selection	long*	O	m	Option base zéro choisie (voir les paramètres d'option).
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long select;
long step;
select = 0;
step = 0;
do
{
    switch (step)
    {
        case 0:
            MenuDisplay(ITEM_ID(wizard_step0), "NEXT", &select);
            step++;
            break;
        case 1:
            MenuDisplay(ITEM_ID(wizard_step1), "BACK;NEXT", &select);
            if (select == 0)
                step = 0;
            if (select == 1)
                step = 2;
            break;
        case 2:
            MenuDisplay(ITEM_ID(wizard_step2), "FINISH", &select);
            step++;
            break;
        default:
            break;
    }
} while (step < 3);

```

6.2.34 Builtin PUT_MESSAGE

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message spécifié à l'écran

Le builtin put_message affiche le message spécifié à l'écran. Les éventuelles références de variable incorporées sont développées en valeur actuelle de la variable. Il convient d'afficher les variables ENUMERATED dans leur représentation textuelle lorsqu'elles sont définies. Lorsqu'une valeur ENUMERATED n'est pas définie, il convient qu'elle soit affichée dans sa représentation numérique.

Profils de communication: HART

Les variables BIT_ENUMERATED doivent être affichées en tant qu'entiers au format hexadécimal.

Profils de communication: PB, PN, CS, GPE

Il convient que les variables BIT_ENUMERATED soient affichées dans leur représentation numérique sous forme d'entier hexadécimal ou dans leur représentation textuelle contenant tous les bits définis sur "1".

PUT_MESSAGE n'accepte aucune entrée de l'opérateur et l'utilisateur n'a pas la possibilité d'interrompre la méthode.

L'exécution de la méthode se poursuit immédiatement, c'est-à-dire qu'elle n'est pas bloquée. La chaîne de message peut contenir des références incorporées à toute combinaison de variables de méthode et de valeurs de VARIABLE. Il convient que le message reste disponible pour examen au moins jusqu'à ce qu'un appel ultérieur remplace potentiellement le message par un nouveau message ou que la méthode se termine.

Les références aux variables peuvent être incorporées à la chaîne de message, ce qui permet d'afficher leur valeur en utilisant la syntaxe indiquée en 6.2.40.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

PUT_MESSAGE (message)

Les éléments lexicaux du builtin PUT_MESSAGE sont présentés dans le Tableau 43.

Tableau 43 – Builtin PUT_MESSAGE

Nom du paramètre	Type	Direction	Utilisation	Description
message	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	VOID	—	—	—

Exemple

```
int sum;
PUT_MESSAGE("The sum is: %[5.2f]{sum}");
```

L'exemple affiche la valeur de la variable "sum".

```
PUT_MESSAGE("The loop current is %[4.2f,D]{loop_current} mA");
```

L'exemple affiche et met constamment à jour la valeur de la variable "loop_current".

```
PUT_MESSAGE("%[L]{flow_value} = %{flow_value}");
```

L'exemple affiche le libellé et la valeur de la variable "flow_value".

6.2.35 Builtin put_message

Profils de communication: CS, GPE, HART, PB, PN

Objet: Afficher le message spécifié à l'écran

Le builtin put_message affiche le message spécifié à l'écran. Les éventuelles références de variable incorporées sont développées en valeur actuelle de la variable. Il convient d'afficher les variables ENUMERATED dans leur représentation textuelle lorsqu'elles sont définies. Lorsqu'une valeur ENUMERATED n'est pas définie, il convient qu'elle soit affichée dans sa représentation numérique.

Profils de communication: HART

Les variables BIT_ENUMERATED doivent être affichées en tant qu'entiers au format hexadécimal.

Profils de communication: PB, PN, CS, GPE

Il convient que les variables BIT_ENUMERATED soient affichées dans leur représentation numérique sous forme d'entier hexadécimal ou dans leur représentation textuelle contenant tous les bits définis sur "1".

put_message n'accepte aucune entrée de l'opérateur et l'utilisateur n'a pas la possibilité d'interrompre la méthode. L'exécution de la méthode se poursuit immédiatement, c'est-à-dire qu'elle n'est pas bloquée. Il convient que le message reste disponible pour examen au moins jusqu'à ce qu'un appel ultérieur remplace potentiellement le message par un nouveau message ou que la méthode se termine.

La chaîne peut accéder à un nombre quelconque d'identifiants, mais les entrées de matrice ciblées doivent contenir des identifiants de variable valides. La chaîne de message peut contenir des références à toute combinaison de variables de méthode et de valeurs de VARIABLE. Les méthodes de variable doivent être référencées par nom et les VARIABLES doivent être référencées par global_var_id.

La syntaxe suivante est utilisée pour référencer une VARIABLE de la liste des VARIABLES donnée dans le paramètre global_var_ids:

```
%{global-var-ids-array-index}
```

ou

```
%[format]{global-var-ids-array-index}
```

Le format facultatif prend en charge toutes les chaînes de format fprintf décrites dans le langage de programmation C (voir l'ISO/IEC 9899:1999), plus les ajouts présentés dans le Tableau 49. Si la chaîne de format n'est pas incluse, le formatage par défaut spécifié dans l'IEC 61804-3 doit être utilisé.

NOTE 1 Même si cela n'est pas recommandé pour les nouvelles DDs, put_message (minuscule) est pris en charge (à des fins de rétrocompatibilité).

NOTE 2 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 3 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
put_message(message, global_var_ids)
```

Les éléments lexicaux du builtin put_message sont présentés dans le Tableau 44.

Tableau 44 – Builtin put_message

Nom du paramètre	Type	Direction	Utilisation	Description
message	DD_STRING	l	m	Message à afficher. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	int[]	l	m	Matrice qui contient des identifiants uniques d'instance de VARIABLE.
<return>	VOID	—	—	—

Exemple

```
int ids[2];
ids[0] = VARID(pressure_units);
ids[1] = VARID(pressure_value);
put_message("Pressure value = {%1} {%0}",ids);
```

L'exemple obtient l'ID des variables "pressure_value" et "pressure_units" et le transmet au builtin put_message pour affichage.

6.2.36 Builtin SELECT_FROM_LIST

Profils de communication: CS, GPE, HART, PB, PN

Objet: Permettre à l'utilisateur de choisir l'une des multiples options

Le builtin SELECT_FROM_LIST affiche une invite et une liste d'options fournie pour permettre le choix de l'utilisateur. Les options sont transmises à la fonction sous la forme d'une chaîne unique délimitée par des points-virgules entre les options. Lorsqu'un choix est effectué, la position de cette option dans la liste d'options est retournée. La position dans la liste est à base zéro. Par exemple, si la liste fournie contient deux éléments et que le premier élément est choisi, cette fonction retourne un 0. Si le deuxième élément est choisi, un 1 est retourné. La liste de sélection doit comporter au moins deux options.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
SELECT_FROM_LIST(prompt, option_list)
```

Les éléments lexicaux du builtin SELECT_FROM_LIST sont présentés dans le Tableau 45.

Tableau 45 – Builtin SELECT_FROM_LIST

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.40 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
option_list	DD_STRING	I	m	Chaîne d'option, constituée de texte délimité, qui définit les éléments de menu. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	int	O	m	Résultat de la sélection sous forme de position dans la liste d'options (base zéro).

Exemple

```
int result;
result = SELECT_FROM_LIST("Is this correct?", "Yes;No");
if (result == 0) { ... } // user selected "yes"
else { ... }
```

Cette fonction affiche l'invite "Is this correct?" et les deux options "Yes" et "No". Si "Yes" est choisi, un 0 est retourné et le code de l'instruction if est exécuté. Si "No" est choisi, un 1 est retourné et le code de l'instruction else est exécuté.

6.2.37 Builtin select_from_list

Profils de communication: CS, GPE, HART, PB, PN

Objet: Permettre à l'utilisateur de choisir l'une des multiples options

Le builtin select_from_list affiche une invite et une liste d'options fournie pour permettre le choix de l'utilisateur. Lorsqu'un choix est effectué, la position de cette option dans la liste d'options est retournée. La position dans la liste est à base zéro. Par exemple, si la liste fournie contient deux éléments et que le premier élément est choisi, cette fonction retourne un 0. Si le deuxième élément est choisi, un 1 est retourné.

L'utilisateur a la possibilité d'interrompre la méthode et son exécution est stoppée dans l'attente d'un retour de la fonction.

Les options sont transmises à la fonction sous la forme d'une chaîne unique délimitée par des points-virgules entre les options. La liste de sélection doit comporter au moins deux options.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

NOTE 3 Les nouvelles descriptions de produits peuvent utiliser SELECT_FROM_LIST, qui prend en charge le référencement direct des variables de méthode et des VARIABLES. La version en minuscules de cette fonction est conservée à des fins de rétrocompatibilité uniquement.

Structure lexicale

```
select_from_list(prompt, global_var_ids, option_list)
```

Les éléments lexicaux du builtin select_from_list sont présentés dans le Tableau 46.

Tableau 46 – Builtin select_from_list

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.35 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
global_var_ids	Int[]	I	m	Matrice qui contient des identifiants uniques d'instances de VARIABLE.
option_list	DD_STRING	I	m	Chaîne d'option, constituée de texte délimité par des points-virgules, qui définit les éléments de menu. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	int	O	m	Résultat de la sélection sous forme de position (base zéro).

6.2.38 Builtin select_from_menu

Profils de communication: FF, ISA100

Objet: Afficher un menu de choix et permettre à l'utilisateur d'effectuer une sélection

Le builtin select_from_menu affiche une liste d'options ou d'éléments de menu pour permettre le choix de l'utilisateur et extrait le numéro de sélection. Lorsque l'utilisateur a fait un choix, celui-ci est retourné à la méthode appelante. Les numéros de sélection commencent à 1 et augmentent jusqu'au nombre d'options spécifié par les options de paramètre.

La chaîne d'options définit le menu dans lequel l'utilisateur effectue un choix. Il s'agit d'une chaîne constituée des options de sélection, chaque option étant séparée des autres options par un point-virgule. Des codes de langue peuvent être incorporés dans chaque option. Le premier élément de menu doit être celui par défaut, c'est-à-dire présélectionné.

L'application affiche les chaînes conformément aux lignes directrices suivantes:

- les éléments de menu apparaissent à l'écran dans l'ordre dans lequel ils sont répertoriés dans la chaîne;
- la chaîne d'invite apparaît en haut de l'affichage du menu à l'écran;
- la chaîne d'invite n'est pas sélectionnable;
- le menu est présenté verticalement.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
select_from_menu(prompt, ids, indices, id_count, options, selection)
```

Les éléments lexicaux du builtin select_from_menu sont présentés dans le Tableau 47.

Tableau 47 – Builtin select_from_menu

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou indices des VARIABLES utilisées dans le message d'invite.
id_count	long	I	m	Nombre d'éléments REFERENCE_ARRAY dans le REFERENCE_ARRAY pour les ID et les indices. Défini sur zéro si le REFERENCE_ARRAY pour les ID et les indices est zéro.
options	DD_STRING	I	m	Chaîne d'option, constituée de texte délimité, qui définit les éléments de menu. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
selection	long	O	m	Résultat de la sélection sous forme de position (base zéro).
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
unsigned long paramIds[1];
unsigned long memberIds[1];
int choice;
float new_value;

status = select_from_menu("en|Is limit correct?|de|Grenze richtig?",
paramIds, memberIds, 0, "en|Yes|de|Ja;en|No|de|Nein", &choice);

switch (choice)
{
case 1:
    status = put_float_value(MEMBER_ID(HI_LIM), new_value);
    status = send_all_values();
    break;

default:
    break;
}

```

6.2.39 Builtin select_from_menu2

Profils de communication: FF, ISA100

Objet: Afficher un menu de choix et permettre à l'utilisateur d'effectuer une sélection

Le builtin select_from_menu2 affiche une liste d'options ou d'éléments de menu pour permettre le choix de l'utilisateur et extrait le numéro de sélection. Lorsque l'utilisateur a fait un choix, celui-ci est retourné à la méthode appelante. Les numéros de sélection commencent à 1 et augmentent jusqu'au nombre d'options spécifié par les options de paramètre.

Les paramètres `blkIds`, `blk_nums`, `ids`, `indices` et `id_count` spécifient collectivement les valeurs à incorporer dans l'invite.

Le paramètre `blk_ids` spécifie les ID d'éléments des blocs auxquels accéder, tandis que le paramètre `blk_nums` spécifie les indices relatifs des blocs auxquels accéder.

Si un élément du paramètre `ids` fait référence à une variable, l'élément correspondant du paramètre `indices` est ignoré. Si un élément du paramètre `ids` fait référence à un enregistrement, l'élément correspondant du paramètre `indices` spécifie un membre de l'enregistrement. Si un élément du paramètre `ids` fait référence à une matrice, l'élément correspondant du paramètre `indices` spécifie un élément de la matrice.

Les éléments des paramètres `blk_ids`, `ids` et `indices` sont généralement spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Les éléments du paramètre `blk_nums` sont généralement spécifiés à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Les paramètres `blk_ids`, `blk_nums` et `ids` peuvent être transmis en tant que valeur NULL si aucune valeur n'est incorporée dans l'invite. Si l'un de ces paramètres est transmis en tant que valeur NULL, ils doivent tous être transmis en tant que valeur NULL.

Le paramètre `indices` peut être transmis en tant que valeur NULL si le paramètre `ids` référence uniquement des variables, c'est-à-dire s'il ne référence ni enregistrements ni matrices.

Le paramètre `id_count` doit être transmis en tant que valeur zéro lorsque les paramètres `blk_ids`, `blk_nums` et `ids` sont transmis en tant que valeur NULL.

La chaîne d'options définit le menu dans lequel l'utilisateur effectue un choix. Il s'agit d'une chaîne constituée des options de sélection, chaque option étant séparée des autres options par un point-virgule. Des codes de langue peuvent être incorporés dans chaque option. Le premier élément de menu doit être celui par défaut, c'est-à-dire présélectionné.

L'application affiche les chaînes conformément aux lignes directrices suivantes:

- les éléments de menu apparaissent à l'écran dans l'ordre dans lequel ils sont répertoriés dans la chaîne;
- la chaîne d'invite apparaît en haut de l'affichage du menu à l'écran;
- la chaîne d'invite n'est pas sélectionnable;
- le menu est présenté verticalement.

NOTE 1 Il n'est pas garanti que les messages précédemment affichés restent à l'écran.

NOTE 2 Certaines applications EDD effacent l'affichage à chaque appel de cette fonction.

Structure lexicale

```
select_from_menu2(prompt, blk_ids, blk_nums, ids, indices, id_count,  
options, selection)
```

Les éléments lexicaux du builtin `select_from_menu2` sont présentés dans le Tableau 48.

Tableau 48 – Builtin select_from_menu2

Nom du paramètre	Type	Direction	Utilisation	Description
prompt	DD_STRING	I	m	Un message à afficher (voir 6.2.6 pour la syntaxe). Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
blk_ids	unsigned long[]	I	m	ID d'éléments des instances de BLOCK_A utilisées dans le message d'invite.
blk_nums	unsigned long[]	I	m	Numéros d'occurrence des instances de BLOCK_A utilisées dans le message d'invite.
ids	unsigned long[]	I	m	ID d'éléments des VARIABLES utilisées dans le message d'invite.
indices	unsigned long[]	I	m	ID de membres ou indices des VARIABLES utilisées dans le message d'invite.
id_count	long	I	m	Nombre d'éléments REFERENCE_ARRAY dans le REFERENCE_ARRAY pour les ID et les indices. Défini sur zéro si le REFERENCE_ARRAY pour les ID et les indices est zéro.
options	DD_STRING	I	m	Chaîne d'option, constituée de texte délimité, qui définit les éléments de menu. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
selection	long	O	m	Résultat de la sélection sous forme de position (base zéro).
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
unsigned long blk_ids[2];
unsigned long blk_nums[2];
unsigned long ids[2];
unsigned long indices[2];
long choice;

blk_ids[0] = ITEM_ID(__analog_input_block);
blk_nums[0] = 0;
ids[0] = MEMBER_ID(MODE_BLK);
indices[0] = MEMBER_ID(ACTUAL);

blk_ids[1] = ITEM_ID(__analog_input_block);
blk_nums[1] = 1;
ids[1] = MEMBER_ID(MODE_BLK);
indices[1] = MEMBER_ID(ACTUAL);

status = select_from_menu2("AI1 Mode: %{0}, AI2 Mode: %{1}\nIs this
correct?", blk_ids, blk_nums, ids, indices, 2, "Yes;No" &choice);

```

6.2.40 Formats de chaîne d'invite

Les références à des VARIABLES peuvent être incorporées dans la chaîne d'invite, ce qui permet d'afficher leur valeur en utilisant la syntaxe suivante:

```
%{variable-reference}
```

ou

```
%[format]{variable-reference}
```

La référence de variable est l'identifiant d'une variable de méthode ou d'une VARIABLE. Variable de méthode et VARIABLES peuvent être mélangées dans le même message. Le format facultatif prend en charge toutes les chaînes de format fprintf décrites dans le langage de programmation C (voir l'ISO//IEC 9899:1999), plus les ajouts présentés dans le Tableau 49. Si la chaîne de format n'est pas incluse, le formatage par défaut spécifié dans l'IEC 61804-3 doit être utilisé.

Tableau 49 – Options de format

Format	Description
D	L'élément référencé est de CLASS DYNAMIC et l'affichage doit être mis à jour de manière continue tandis que la fonction builtin bloque l'exécution de la méthode. Les éléments de CLASS DYNAMIC référencés sans l'option D ne doivent pas être mis à jour de manière dynamique. Les éléments qui ne sont pas de CLASS DYNAMIC référencés avec l'option D ne doivent pas être affichés de manière dynamique. Les builtins d'interface utilisateur non bloquants (par exemple, PUT_MESSAGE et put_message) affichent une seule fois un instantané de la valeur dynamique. Si un format d'affichage est donné en plus du spécificateur "D", ce dernier doit être séparé du spécificateur de format par une virgule. Exemple: PUT_MESSAGE ("La variable avec le libellé '%[L]{myVar}' a la valeur dynamique: %[3.1f,D]{myVar}");
L	Le LABEL de l'élément référencé doit être affiché. Exemple: PUT_MESSAGE ("Le libellé de ma variable est: %[L]{myVar}");
U	L'unité de l'élément référencé doit être affichée. Exemple: PUT_MESSAGE ("Les unités de ma variable sont: %[U]{myVar}");

6.3 Builtins de communication

6.3.1 Builtin abort_on_all_comm_errors

Profils de communication: FF, ISA100

Objet: Définir l'interruption pour toutes les erreurs de communication

Le builtin abort_on_all_comm_errors spécifie l'interruption de la méthode en cas de réception d'erreur de communication.

Lorsqu'une méthode est lancée, elle n'est pas interrompue sur une erreur de communication tant qu'abort_on_all_comm_errors n'est pas appelé. Cela évite à une méthode de devoir gérer des codes d'erreur de communication en interrompant la méthode si elle reçoit une erreur de communication.

Après cet appel, tout appel de builtin qui effectue une demande de communication qui retourne une erreur conduit à une interruption de la méthode et au lancement de l'exécution de sa liste de méthodes d'interruption. Les codes d'erreur de communication sont différents de zéro et dépendent d'une application EDD (chaque application EDD définit ses propres codes).

Structure lexicale

```
abort_on_all_comm_errors (VOID)
```


L'élément lexical du builtin `abort_on_all_comm_errors` est présenté dans le Tableau 50.

Tableau 50 – Builtin `abort_on_all_comm_errors`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
abort_on_all_comm_errors();
```

6.3.2 Builtin `ABORT_ON_ALL_COMM_STATUS`

Profils de communication: HART

Objet: Définir l'interruption pour tous les statuts de communication

Le builtin `ABORT_ON_ALL_COMM_STATUS` définit tous les bits dans le masque d'interruption de statut de communication. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de statut de communication. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini sur 1.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

```
ABORT_ON_ALL_COMM_STATUS(VOID)
```

L'élément lexical du builtin `ABORT_ON_ALL_COMM_STATUS` est présenté dans le Tableau 51.

Tableau 51 – Builtin `ABORT_ON_ALL_COMM_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.3 Builtin `ABORT_ON_ALL_DEVICE_STATUS`

Profils de communication: HART

Objet: Définir l'interruption pour tous les statuts de dispositif

Le builtin `ABORT_ON_ALL_DEVICE_STATUS` définit tous les bits dans le masque d'interruption de statut du dispositif. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de statut du dispositif différente de zéro. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le premier octet retourné indique un statut de code de réponse lorsque le bit de poids fort (bit 7) a la valeur 0. Le deuxième octet retourné, statut du dispositif, est significatif uniquement si le premier retourne un statut de code de réponse. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`ABORT_ON_ALL_DEVICE_STATUS (VOID)`

L'élément lexical du builtin `ABORT_ON_ALL_DEVICE_STATUS` est présenté dans le Tableau 52.

Tableau 52 – Builtin `ABORT_ON_ALL_DEVICE_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.4 Builtin `ABORT_ON_ALL_RESPONSE_CODES`

Profils de communication: HART

Objet: Définir l'interruption pour tous les codes de réponse

Le builtin `ABORT_ON_ALL_RESPONSE_CODES` définit tous les bits dans le masque d'interruption de code de réponse. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de code de réponse différente de zéro.

Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction de commande d'envoi pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

NOTE Les réponses et activités retardées sont des codes de niveau Data-Link qui ne sont pas retournés aux méthodes, c'est-à-dire que les niveaux inférieurs de la pile de communication ou les I/O distantes gèrent/interagissent avec les codes de réponse. En fonction du dispositif terminal avec lequel la communication est établie, une `METHOD` peut voir un `DR_DEAD` (échec du DR).

Structure lexicale

`ABORT_ON_ALL_RESPONSE_CODES (VOID)`

L'élément lexical du builtin `ABORT_ON_ALL_RESPONSE_CODES` est présenté dans le Tableau 53.

Tableau 53 – Builtin ABORT_ON_ALL_RESPONSE_CODES

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.5 Builtin abort_on_all_response_codes

Profils de communication: FF, ISA100

Objet: Définir l'interruption pour tous les codes de réponse

Le builtin abort_on_all_response_codes conduit à l'interruption de la méthode si celle-ci reçoit un code de réponse différent de zéro, après l'appel d'un builtin qui conduit à une demande à un dispositif. Ce builtin évite à une méthode de devoir gérer des codes de réponse d'erreur. Lorsqu'une méthode est lancée, elle n'est pas interrompue sur un code de réponse tant que ce builtin n'est pas appelé. Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application. Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

Structure lexicale

abort_on_all_response_codes(VOID)

L'élément lexical du builtin abort_on_all_response_codes est présenté dans le Tableau 54.

Tableau 54 – Builtin abort_on_all_response_codes

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
abort_on_all_response_codes();
```

6.3.6 Builtin ABORT_ON_COMM_ERROR

Profils de communication: HART

Objet: Interrompre si une erreur de communication est détectée par le maître ou l'esclave

Le builtin ABORT_ON_COMM_ERROR définit le masque d'interruption sur erreur de communication, de sorte que la méthode soit interrompue si une erreur de communication est détectée dans le dispositif ou l'application EDD lors de l'envoi d'une commande.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

ABORT_ON_COMM_ERROR(VOID)

L'élément lexical du builtin ABORT_ON_COMM_ERROR est présenté dans le Tableau 55.

Tableau 55 – Builtin ABORT_ON_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.7 Builtin abort_on_comm_error

Profils de communication: FF, ISA100

Objet: Définir le masque utilisé pour interrompre la méthode lorsqu'une erreur de communication est détectée par le dispositif esclave

Le builtin abort_on_comm_error conduit à l'interruption de la méthode si celle-ci reçoit une erreur de communication spécifique.

Lorsqu'une méthode est lancée, elle n'est pas interrompue sur une erreur de communication spécifique tant que le builtin abort_on_comm_error n'est pas appelé.

NOTE Une erreur de communication égale à zéro n'est pas assimilée à une erreur de communication.

Ce builtin permet à l'auteur de la méthode de choisir les erreurs de communication qu'il convient que la méthode gère, et il évite à une méthode de devoir gérer un code d'erreur de communication spécifique.

Les codes d'erreur de communication sont définis par les consortiums. Leurs chaînes associées peuvent être définies dans un dictionnaire de texte.

Structure lexicale

```
abort_on_comm_error(error)
```

Les éléments lexicaux du builtin abort_on_comm_error sont présentés dans le Tableau 56.

Tableau 56 – Builtin abort_on_comm_error

Nom du paramètre	Type	Direction	Utilisation	Description
error	unsigned long	I	m	Code d'erreur de communication.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = abort_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.8 Builtin ABORT_ON_COMM_STATUS

Profils de communication: HART

Objet: Définir le masque utilisé pour interrompre la méthode lorsqu'une erreur de communication est détectée par le dispositif esclave

Le builtin ABORT_ON_COMM_STATUS définit le ou les bits corrects dans le masque d'interruption de statut de communication, de sorte que la valeur comm_status spécifiée conduise à l'interruption de la méthode. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini. La valeur de comm_status peut être définie sur toute valeur de 0x00 à 0x7F.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`ABORT_ON_COMM_STATUS(comm_status)`

Les éléments lexicaux du builtin `ABORT_ON_COMM_STATUS` sont présentés dans le Tableau 57.

Tableau 57 – Builtin `ABORT_ON_COMM_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>comm_status</code>	unsigned char	l	m	Masque binaire pour le statut de communication.
<return>	VOID	—	—	—

6.3.9 Builtin `ABORT_ON_DEVICE_STATUS`

Profils de communication: HART

Objet: Définir le masque d'interruption de statut du dispositif

Le builtin `ABORT_ON_DEVICE_STATUS` définit les bits corrects dans le masque d'interruption de statut du dispositif, de sorte que la valeur de statut du dispositif spécifiée conduise à l'interruption de la méthode. Les deux premiers octets de chaque réponse de l'esclave contiennent des informations de statut. Le premier octet retourné indique un statut de code de réponse lorsque le bit de poids fort (bit 7) a la valeur 0. Le deuxième octet retourné, statut du dispositif, est significatif uniquement si un statut de code de réponse est retourné par le premier octet. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction. La valeur de `device_status` peut être définie de manière à conduire à l'interruption de la méthode actuelle sur toute combinaison de bits de statut du dispositif. Cela s'obtient en créant un masque associé soumis à un AND sur le statut du dispositif. La valeur de `device_status` peut être définie sur toute valeur de 0x00 à 0x7F.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction de commande d'envoi pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`ABORT_ON_DEVICE_STATUS(device_status)`

Les éléments lexicaux du builtin `ABORT_ON_DEVICE_STATUS` sont présentés dans le Tableau 58.

Tableau 58 – Builtin ABORT_ON_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
device_status	unsigned int	l	m	Masque binaire pour le statut du dispositif.
<return>	VOID	—	—	—

6.3.10 Builtin ABORT_ON_NO_DEVICE (déconseillé)**Profils de communication:****Objet: Interrompre si aucun dispositif n'est détecté**

Le builtin ABORT_ON_NO_DEVICE définit le masque d'interruption en l'absence de dispositif, de sorte que la méthode soit interrompue si aucun dispositif n'est détecté lors de l'envoi d'une commande.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction de commande d'envoi pour la mise en œuvre des masques. Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

ABORT_ON_NO_DEVICE (VOID)

L'élément lexical du builtin ABORT_ON_NO_DEVICE est présenté dans le Tableau 59.

Tableau 59 – Builtin ABORT_ON_NO_DEVICE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.11 Builtin ABORT_ON_RESPONSE_CODE**Profils de communication: HART****Objet: Définir le masque d'interruption de code de réponse**

Le builtin ABORT_ON_RESPONSE_CODE définit le ou les bits corrects dans le masque d'interruption de code de réponse, de sorte que la valeur de code de réponse spécifiée conduise à l'interruption de la méthode. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0. La valeur de response_code peut être définie de manière à conduire à une interruption de la méthode sur tout code de réponse. La valeur de response_code peut être définie sur toute valeur de 0 à 127.

Pour conduire à l'interruption de la commande HART actuelle sur plusieurs codes de réponse différents, ce builtin doit être appelé une fois pour chaque code de réponse à interrompre.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle.

La liste de tous les masques disponibles, avec leurs valeurs par défaut, est présentée dans le Tableau 60:

Tableau 60 – Masques d'interruption et de relance disponibles

Masques	Valeurs par défaut
response_code_abort_mask	ACCESS_RESTRICTED CMD_NOT_IMPLEMENTED
response_code_retry_mask	Tous les bits définis sur 0
cmd_48_response_code_abort_mask	ACCESS_RESTRICTED CMD_NOT_IMPLEMENTED
cmd_48_response_code_retry_mask	Tous les bits définis sur 0
device_status_abort_mask	DEVICE_MALFUNCTION
device_status_retry_mask	Tous les bits définis sur 0
cmd_48_device_status_abort_mask	DEVICE_MALFUNCTION
cmd_48_device_status_retry_mask	Tous les bits définis sur 0
comm_status_abort_mask	Tous les bits définis sur 0
comm_status_retry_mask	Tous les bits définis sur 1
cmd_48_comm_status_abort_mask	Tous les bits définis sur 0
cmd_48_comm_status_retry_mask	Tous les bits définis sur 1
no_device_abort_mask	Tous les bits définis sur 1
comm_error_abort_mask	Tous les bits définis sur 1
comm_error_retry_mask	Tous les bits définis sur 0
cmd_48_no_device_abort_mask	Tous les bits définis sur 1
cmd_48_comm_error_abort_mask	Tous les bits définis sur 1
cmd_48_comm_error_retry_mask	Tous les bits définis sur 0
cmd_48_data_abort_mask	Tous les bits définis sur 0
cmd_48_data_retry_mask	Tous les bits définis sur 0

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

ABORT_ON_RESPONSE_CODE(response_code)

Les éléments lexicaux du builtin ABORT_ON_RESPONSE_CODE sont présentés dans le Tableau 61.

Tableau 61 – Builtin ABORT_ON_RESPONSE_CODE

Nom du paramètre	Type	Direction	Utilisation	Description
response_code	int	l	m	Code de réponse qu'il convient d'ajouter au masque d'interruption.
<return>	VOID	—	—	—

6.3.12 Builtin abort_on_response_code

Profils de communication: FF, ISA100

Objet: Définir le masque d'interruption de code de réponse

Le builtin abort_on_response_code conduit à l'interruption de la méthode après réception du code de réponse différent de zéro spécifié. Ce builtin évite à une méthode de devoir gérer un code de réponse différent de zéro spécifique.

Lorsqu'une méthode est lancée, elle n'est pas interrompue sur un code de réponse spécifique tant que ce builtin n'est pas appelé. Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application.

NOTE Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque type de code de réponse est défini dans l'EDD.

Structure lexicale

abort_on_response_code (code)

Les éléments lexicaux du builtin abort_on_response_code sont présentés dans le Tableau 62.

Tableau 62 – Builtin abort_on_response_code

Nom du paramètre	Type	Direction	Utilisation	Description
code	unsigned long	I	m	Masque binaire pour le code de réponse.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = abort_on_response_code(RC_INVALID_ENTRY);
```

6.3.13 Builtin ext_send_command

Profils de communication: HART

Objet: Commande d'envoi

Le builtin ext_send_command a la même fonctionnalité que la fonction send_command, sauf que le statut de commande et les octets de données retournés par le dispositif avec la commande 48 sont également retournés.

La fonction appelante est responsable de l'allocation des matrices pour les données à retourner. Les VARIABLES cmd_status et more_data_status sont des pointeurs vers des matrices de 3 octets, et la VARIABLE more_data_info est un pointeur vers une matrice dont la taille est égale au nombre d'octets de données qui peuvent être retournés par le dispositif dans la commande 48 (25 au plus). Si le bit de statut indiquant que des données supplémentaires sont disponibles est retourné (classe de statut MORE), une commande 48 est automatiquement émise. Si la commande 48 n'a pas été émise, les octets more_data_status et more_data_info sont définis sur zéro.

Les octets de la matrice `cmd_status` et les octets de statut de la commande 48 (si elle a été envoyée) sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. S'il existe une correspondance, la commande est relancée (la commande spécifiée ou la commande 48). Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Lorsqu'une commande 48 est envoyée indirectement en tant que résultat de réponse de statut du dispositif, les valeurs du masque xxx de la commande 48 définies à l'aide des builtins `XMTR_xxx` sont utilisées pour contrôler les interruptions et relances pour la commande 48. Les autres valeurs de masque et builtins associés sont utilisés pour contrôler les interruptions et relances de la commande directement envoyée (y compris la commande 48 si elle a été directement envoyée).

Structure lexicale

`ext_send_command(cmd_number, cmd_status, more_data_status, more_data_info)`

Les éléments lexicaux du builtin `ext_send_command` sont présentés dans le Tableau 63.

Tableau 63 – Builtin `ext_send_command`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la COMMAND.
<code>cmd_status</code>	char[]	I	m	Statut de la COMMAND.
<code>more_data_status</code>	char[]	I	m	Spécifie si des données de statut supplémentaires sont disponibles.
<code>more_data_info</code>	char[]	I	m	Spécifie si des informations supplémentaires sont disponibles.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. Retourne <code>BI_SUCCESS</code> si la commande a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

6.3.14 Builtin `ext_send_command_trans`

Profils de communication: HART

Objet: Envoyer la transaction de commande spécifiée

Le builtin `ext_send_command_trans` envoie la commande avec la transaction spécifiée au dispositif. Cette fonction doit être utilisée pour envoyer des commandes qui ont été définies avec des transactions multiples.

Le builtin `ext_send_command_trans` a la même fonctionnalité que la fonction `send_command_trans`, sauf que le statut de commande et les octets de données retournés par le dispositif avec la commande 48 sont également retournés.

La fonction appelante est responsable de l'allocation des matrices pour les données à retourner. Les variables `cmd_status` et `more_data_status` sont des pointeurs vers des matrices de 3 octets, et la variable `more_data_info` est un pointeur vers une matrice dont la taille est égale au nombre d'octets de données qui peuvent être retournés par le dispositif dans la commande 48 (25 au plus). Si la commande 48 n'a pas été émise, les octets `more_data_status` et `more_data_info` sont définis sur zéro.

Les octets de la matrice `cmd_status` et les octets de statut de la commande 48 (si elle a été envoyée) sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. S'il existe une correspondance, la commande est relancée (la commande spécifiée ou la commande 48). Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Lorsqu'une commande 48 est envoyée indirectement en tant que résultat de réponse de statut du dispositif, les valeurs du masque xxx de la commande 48 définies à l'aide des builtins `XMTR_xxx` sont utilisées pour contrôler les interruptions et relances pour la commande 48. Les autres valeurs de masque et builtins associés sont utilisés pour contrôler les interruptions et relances de la commande directement envoyée (y compris la commande 48 si elle a été directement envoyée).

Structure lexicale

```
ext_send_command_trans(cmd_number, transaction, cmd_status,
more_data_status, more_data_info)
```

Les éléments lexicaux du builtin `ext_send_command_trans` sont présentés dans le Tableau 64.

Tableau 64 – Builtin `ext_send_command_trans`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la COMMAND.
<code>transaction</code>	int	I	m	Numéro de la TRANSACTION.
<code>cmd_status</code>	char[]	I	m	Statut de la COMMAND.
<code>more_data_status</code>	char[]	I	m	Spécifie si des données de statut supplémentaires sont disponibles.
<code>more_data_info</code>	char[]	I	m	Spécifie si des informations supplémentaires sont disponibles.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. Retourne <code>BI_SUCCESS</code> si la commande a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

6.3.15 Builtin `fail_on_all_comm_errors`

Profils de communication: FF, ISA100

Objet: Renvoyer le contrôle à la méthode en cas d'erreurs de communication

Le builtin `fail_on_all_comm_errors` déclenche le retour du contrôle à la méthode appelante lorsqu'une erreur de communication est détectée par le builtin appelé au lieu d'interrompre la méthode ou de relancer la demande de communication.

Au lieu d'interrompre la méthode ou de tenter de relancer une demande de communication en échec lorsqu'une erreur de communication se produit, le builtin `fail_on_all_comm_errors` initialise les builtins pour envoyer un code d'erreur de communication à la méthode.

Lorsqu'une erreur de communication se produit, le builtin effectuant une demande de communication retourne l'erreur `BLTIN_FAIL_COMM_ERROR`. La méthode peut appeler le builtin `get_comm_error` puis `get_comm_error_string` pour obtenir des informations supplémentaires concernant l'erreur et ensuite afficher la chaîne d'erreur, le cas échéant.

Structure lexicale

`fail_on_all_comm_errors(VOID)`

L'élément lexical du builtin `fail_on_all_comm_errors` est présenté dans le Tableau 65.

Tableau 65 – Builtin `fail_on_all_comm_errors`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
fail_on_all_comm_errors();
```

6.3.16 Builtin `fail_on_all_response_codes`

Profils de communication: FF, ISA100

Objet: Renvoyer le contrôle à la méthode en cas de codes de réponse

Le builtin `fail_on_all_response_codes` déclenche le retour du contrôle à la méthode appelante lorsqu'un code de réponse différent de zéro est détecté par le builtin appelé au lieu d'interrompre la méthode ou de la relancer.

Lorsqu'une erreur de code de réponse se produit, tout builtin effectuant une demande retourne l'erreur `BLTIN_FAIL_RESPONSE_CODE`. La méthode peut appeler le builtin `get_comm_error` puis `get_comm_error_string` pour obtenir des informations supplémentaires concernant l'erreur, le cas échéant.

Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application.

NOTE Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

Structure lexicale

`fail_on_all_response_codes(VOID)`

L'élément lexical du builtin `fail_on_all_response_codes` est présenté dans le Tableau 66.

Tableau 66 – Builtin `fail_on_all_response_codes`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
fail_on_all_response_codes();
```

6.3.17 Builtin fail_on_comm_error

Profils de communication: FF, ISA100

Objet: Renvoyer le contrôle à la méthode en cas d'erreur de communication spécifiée

Le builtin fail_on_comm_error déclenche le retour du contrôle à la méthode appelante lorsqu'une erreur de communication spécifiée est détectée par le builtin appelé au lieu d'interrompre la méthode ou de la relancer.

Lorsque l'erreur de communication spécifiée se produit, tout builtin effectuant une demande de communication retourne l'erreur BLTIN_FAIL_COMM_ERROR. La méthode peut appeler le builtin get_comm_error puis get_comm_error_string pour obtenir des informations supplémentaires concernant l'erreur et ensuite les afficher, le cas échéant.

Les codes d'erreur de communication sont définis par les consortiums. Leurs chaînes associées peuvent être définies dans un dictionnaire de texte. Une erreur de communication égale à zéro n'est pas assimilée à une erreur de communication.

Structure lexicale

```
fail_on_comm_error(error)
```

Les éléments lexicaux du builtin fail_on_comm_error sont présentés dans le Tableau 67.

Tableau 67 – Builtin fail_on_comm_error

Nom du paramètre	Type	Direction	Utilisation	Description
error	unsigned long	I	m	Code d'erreur de communication.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = fail_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.18 Builtin fail_on_response_code

Profils de communication: FF, ISA100

Objet: Renvoyer le contrôle à la méthode en cas de code de réponse spécifié

Le builtin fail_on_response_code déclenche le retour du contrôle à la méthode appelante lorsqu'un code de réponse différent de zéro est détecté par le builtin appelé au lieu d'interrompre la méthode ou de la relancer.

Lorsque l'erreur de code de réponse spécifiée se produit, tout builtin effectuant une demande retourne l'erreur BLTIN_FAIL_RESPONSE_CODE. La méthode peut appeler le builtin get_comm_error puis get_comm_error_string pour obtenir des informations supplémentaires concernant l'erreur et ensuite les afficher, le cas échéant.

Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application.

NOTE Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

Structure lexicale

`fail_on_response_code(code)`

Les éléments lexicaux du builtin `fail_on_response_code` sont présentés dans le Tableau 68.

Tableau 68 – Builtin `fail_on_response_code`

Nom du paramètre	Type	Direction	Utilisation	Description
code	unsigned long	I	m	Code de réponse.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = fail_on_response_code(RC_INVALID_ENTRY);
```

6.3.19 Builtin `get_more_status`

Profils de communication: HART

Objet: Obtenir les résultats d'une commande 48

Le builtin `get_more_status` transmet une commande 48 au dispositif et retourne le code de réponse, le statut de communication et les octets de statut de commande dans la matrice de statut fournie. Il retourne également les octets de données retournés dans la matrice `more_data_info`. Cette fonction traite les octets de statut/données de la même façon que la commande d'envoi.

La fonction appelante est responsable de l'allocation des matrices pour les données à retourner. Le paramètre `more_data_status` est une matrice de 3 octets et `more_data_info` est une matrice dont la taille est égale au nombre d'octets de données qui peuvent être retournés par le dispositif dans la commande 48 (25 au plus).

Les octets `more_data_status` de la commande 48 sont soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. S'il existe une correspondance, la commande est relancée (la commande spécifiée ou la commande 48). Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Les valeurs du masque xxx de la commande 48 définies à l'aide des builtins `XMTR_xxx` sont utilisées pour contrôler les interruptions et relances pour la commande 48.

Structure lexicale

`get_more_status(more_data_status, more_data_info)`

Les éléments lexicaux du builtin `get_more_status` sont présentés dans le Tableau 69.

Tableau 69 – Builtin get_more_status

Nom du paramètre	Type	Direction	Utilisation	Description
more_data_status	char[]	I	m	Spécifie si des informations supplémentaires sont disponibles.
more_data_info	char[]	I	m	Spécifie si des informations supplémentaires sont disponibles.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. Retourne BI_SUCCESS si la commande a abouti, BI_COMM_ERROR si une erreur s'est produite lors de l'envoi de la commande ou BI_NO_DEVICE si aucun dispositif n'a été détecté.

6.3.20 Builtin IGNORE_ALL_COMM_STATUS

Profils de communication: HART

Objet: Ignorer tous les bits de statut de communication

Le builtin IGNORE_ALL_COMM_STATUS efface tous les bits dans le masque de relance de statut de communication et le masque d'interruption de statut de communication. Cela conduit le système à ignorer tous les bits dans la valeur de statut de communication. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir le builtin send_command pour la mise en œuvre des masques. Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_ALL_COMM_STATUS (VOID)

L'élément lexical du builtin IGNORE_ALL_COMM_STATUS est présenté dans le Tableau 70.

Tableau 70 – Builtin IGNORE_ALL_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.21 Builtin IGNORE_ALL_DEVICE_STATUS

Profils de communication: HART

Objet: Ignorer tous les bits de statut du dispositif

Le builtin IGNORE_ALL_DEVICE_STATUS efface tous les bits dans le masque de relance de statut du dispositif et le masque d'interruption de statut du dispositif. Cela conduit le système à ignorer tous les bits dans l'octet de statut du dispositif. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le premier octet retourné indique un statut de code de réponse lorsque le bit de poids fort (bit 7) a la valeur 0. Le deuxième octet retourné, statut du dispositif, est significatif uniquement si le premier retourne un statut de code de réponse. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir le builtin send_command pour la mise en œuvre des masques. Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_ALL_DEVICE_STATUS (VOID)

L'élément lexical du builtin IGNORE_ALL_DEVICE_STATUS est présenté dans le Tableau 71.

Tableau 71 – Builtin IGNORE_ALL_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.22 Builtin IGNORE_ALL_RESPONSE_CODES

Profils de communication: HART

Objet: Ignorer tous les codes de réponse

Le builtin IGNORE_ALL_RESPONSE_CODES efface tous les bits dans le masque de relance de code de réponse et le masque d'interruption de code de réponse. Cela conduit le système à ignorer toutes les valeurs de code de réponse retournées par le dispositif.

Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir le builtin send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_ALL_RESPONSE_CODES (VOID)

L'élément lexical du builtin IGNORE_ALL_RESPONSE_CODES est présenté dans le Tableau 72.

Tableau 72 – Builtin IGNORE_ALL_RESPONSE_CODES

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.23 Builtin IGNORE_COMM_ERROR

Profils de communication: HART

Objet: Ignorer les erreurs de communication

Le builtin IGNORE_COMM_ERROR définit le masque de relance sur erreur de communication et le masque d'interruption sur erreur de communication, de sorte que toute condition d'erreur de communication dans le dispositif ou l'application EDD soit ignorée lors de l'envoi d'une commande. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le premier octet retourné indique un statut de communication lorsque le bit de poids fort (bit 7) a la valeur 1.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_COMM_ERROR (VOID)

L'élément lexical du builtin IGNORE_COMM_ERROR est présenté dans le Tableau 73.

Tableau 73 – Builtin IGNORE_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.24 Builtin IGNORE_COMM_STATUS

Profils de communication: HART

Objet: Ignorer des bits de statut de communication spécifiques

Le builtin IGNORE_COMM_STATUS efface le ou les bits dans le masque de relance de statut de communication et le masque d'interruption de statut de communication, de sorte que les bits spécifiés dans la valeur comm_status soient ignorés. comm_status est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 1. La valeur de comm_status peut être définie sur toute valeur de 0x00 à 0x7F, où chaque bit correspond à un statut de communication retourné par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`IGNORE_COMM_STATUS(comm_status)`

Les éléments lexicaux du builtin `IGNORE_COMM_STATUS` sont présentés dans le Tableau 74.

Tableau 74 – Builtin `IGNORE_COMM_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>comm_status</code>	<code>int</code>	<code>l</code>	<code>m</code>	Nouveau masque de relance et d'interruption pour ignorer les valeurs.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.25 Builtin `IGNORE_DEVICE_STATUS`

Profils de communication: HART

Objet: Ignorer des bits de statut du dispositif spécifiques

Le builtin `IGNORE_DEVICE_STATUS` efface le ou les bits corrects dans le masque de relance de statut du dispositif et le masque d'interruption de statut du dispositif, de sorte que les bits spécifiés dans l'octet de statut du dispositif soient ignorés. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`IGNORE_DEVICE_STATUS(device_status)`

Les éléments lexicaux du builtin `IGNORE_DEVICE_STATUS` sont présentés dans le Tableau 75.

Tableau 75 – Builtin `IGNORE_DEVICE_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>device_status</code>	<code>int</code>	<code>l</code>	<code>m</code>	Nouveau masque de relance et d'interruption pour ignorer les valeurs.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.26 Builtin IGNORE_NO_DEVICE (déconseillé)

Profils de communication: HART

Objet: Ignorer si aucun dispositif n'est détecté

Le builtin IGNORE_NO_DEVICE efface les bits corrects dans le masque de relance en l'absence de dispositif et le masque d'interruption en l'absence de dispositif pour indiquer qu'il convient d'ignorer la condition d'absence de dispositif lors de l'envoi d'une commande.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_NO_DEVICE (VOID)

L'élément lexical du builtin IGNORE_NO_DEVICE est présenté dans le Tableau 76.

Tableau 76 – Builtin IGNORE_NO_DEVICE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.27 Builtin IGNORE_RESPONSE_CODE

Profils de communication: HART

Objet: Ignorer des codes de réponse spécifiques

Le builtin IGNORE_RESPONSE_CODE efface le bit correct dans le masque de relance de code de réponse et le masque d'interruption de code de réponse, de sorte que la valeur de code de réponse spécifiée soit ignorée. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0. La valeur de response_code peut être définie sur toute valeur de 0 à 127.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

IGNORE_RESPONSE_CODE (response_code)

Les éléments lexicaux du builtin IGNORE_RESPONSE_CODE sont présentés dans le Tableau 77.

Tableau 77 – Builtin IGNORE_RESPONSE_CODE

Nom du paramètre	Type	Direction	Utilisation	Description
response_code	int	I	m	Code de réponse qu'il convient d'ajouter au masque pour ignorer les valeurs.
<return>	VOID	—	—	—

6.3.28 Builtin read_value**Profils de communication: FF, ISA100****Objet: Mettre à jour une valeur de paramètre dans la mémoire cache en la lisant depuis le dispositif**

Le builtin read_value commande la lecture de la valeur du paramètre depuis le dispositif et stocke la valeur dans la mémoire cache.

Le builtin read_value n'effectue aucune action dans une application EDD non en mémoire cache, car une application EDD non en mémoire cache n'a pas de mémoire cache à mettre à jour.

Le builtin read_value fait l'objet des mêmes temporisations de lecture et d'écriture qu'une application qui lit ou écrit une valeur.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la variable en question.

Structure lexicale

```
read_value(id, member_id)
```

Les éléments lexicaux du builtin read_value sont présentés dans le Tableau 78.

Tableau 78 – Builtin read_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = read_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(UNIT_CD));
```

6.3.29 Builtin read_value2**Profils de communication: FF, ISA100****Objet: Lire une variable, un membre d'enregistrement ou un élément de matrice depuis le dispositif**

Le builtin read_value2 commande la lecture de la valeur du paramètre depuis le dispositif et stocke la valeur dans la mémoire cache.

Le builtin `read_value2` n'effectue aucune action dans une application EDD non en mémoire cache, car une application EDD non en mémoire cache n'a pas de mémoire cache à mettre à jour.

Le paramètre `id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une variable simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Si le paramètre `id` référence un enregistrement et que le paramètre `member_id` a la valeur zéro, l'ensemble de l'enregistrement est lu depuis le dispositif. De même, si le paramètre `id` référence une matrice et que le paramètre `member_id` a la valeur zéro, l'ensemble de la matrice est lu depuis le dispositif.

Structure lexicale

```
read_value2(blk_id, blk_num, id, member_id)
```

Les éléments lexicaux du builtin `read_value2` sont présentés dans le Tableau 79.

Tableau 79 – Builtin `read_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
<code>id</code>	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = read_value2(ITEM_ID(__analog_input_block), 0, MEMBER_ID(PV_SCALE),
MEMBER_ID(UNIT_CD));
```

6.3.30 Builtin ReadCommand

Profils de communication: CS, GPE, PB, PN

Objet: Exécuter une commande de lecture

Le builtin `ReadCommand` permet de lire des variables avec une commande spécifiée.

Structure lexicale

```
ReadCommand(command_identifieur)
```

Les éléments lexicaux du builtin ReadCommand sont présentés dans le Tableau 80.

Tableau 80 – Builtin ReadCommand

Nom du paramètre	Type	Direction	Utilisation	Description
command_identifier	reference	I	m	ID de la commande à exécuter.
<return>	long	O	m	Valeur d'erreur spécifique au protocole, dont le codage est défini par le protocole ou dispositif spécifique.

6.3.31 Builtin retry_on_all_comm_errors

Profils de communication: FF, ISA100

Objet: Builtin de relance en cas d'erreurs de communication

Le builtin `retry_on_all_comm_errors` commande la relance de la dernière demande effectuée par une méthode après une erreur de communication. Une demande est relancée trois fois avant qu'elle n'échoue et qu'un retour d'erreur ne soit retourné.

Ce builtin évite à une méthode de devoir relancer explicitement une demande après une erreur de communication.

Les codes d'erreur de communication sont définis dans les profils normalisés et leurs chaînes associées sont définies dans le dictionnaire de texte.

NOTE Une erreur de communication égale à zéro n'est pas assimilée à une erreur de communication.

Structure lexicale

`retry_on_all_comm_errors(VOID)`

L'élément lexical du builtin `retry_on_all_comm_errors` est présenté dans le Tableau 81.

Tableau 81 – Builtin retry_on_all_comm_errors

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
retry_on_all_comm_errors();
```

6.3.32 Builtin RETRY_ON_ALL_COMM_STATUS

Profils de communication: HART

Objet: Définir la relance pour tous les statuts de communication

Le builtin `RETRY_ON_ALL_COMM_STATUS` définit tous les bits dans le masque de relance de statut de communication. Cela conduit le système à relancer la commande actuelle si le dispositif retourne une valeur de statut de communication. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`RETRY_ON_ALL_COMM_STATUS (VOID)`

L'élément lexical du builtin `RETRY_ON_ALL_COMM_STATUS` est présenté dans le Tableau 82.

Tableau 82 – Builtin `RETRY_ON_ALL_COMM_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.33 Builtin `RETRY_ON_ALL_DEVICE_STATUS`

Profils de communication: HART

Objet: Définir la relance pour tous les statuts du dispositif

Le builtin `RETRY_ON_ALL_DEVICE_STATUS` définit tous les bits dans le masque de relance de statut du dispositif. Cela conduit le système à relancer la commande actuelle si le dispositif retourne une valeur de statut du dispositif. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le premier octet retourné indique un statut de code de réponse lorsque le bit de poids fort (bit 7) a la valeur 0. Le deuxième octet retourné, statut du dispositif, est significatif uniquement si le premier retourne un statut de code de réponse. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`RETRY_ON_ALL_DEVICE_STATUS (VOID)`

L'élément lexical du builtin `RETRY_ON_ALL_DEVICE_STATUS` est présenté dans le Tableau 83.

Tableau 83 – Builtin `RETRY_ON_ALL_DEVICE_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.34 Builtin `RETRY_ON_ALL_RESPONSE_CODES`

Profils de communication: HART

Objet: Définir la relance pour tous les codes de réponse

Le builtin `RETRY_ON_ALL_RESPONSE_CODES` définit tous les bits dans le masque de relance de code de réponse. Cela conduit le système à relancer la commande actuelle si le dispositif retourne une valeur de code de réponse différente de zéro.

Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`RETRY_ON_ALL_RESPONSE_CODES` (VOID)

L'élément lexical du builtin `RETRY_ON_ALL_RESPONSE_CODES` est présenté dans le Tableau 84.

Tableau 84 – Builtin `RETRY_ON_ALL_RESPONSE_CODES`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.35 Builtin `retry_on_all_response_codes`

Profils de communication: FF, ISA100

Objet: Définir la relance pour tous les codes de réponse

Le builtin `retry_on_all_response_codes` commande la relance de la dernière demande effectuée par une méthode après un code de réponse quelconque. Une demande est relancée trois fois avant qu'elle n'échoue et qu'un retour d'erreur ne soit retourné. Il s'agit de l'action par défaut effectuée par les builtins lorsqu'une méthode vient d'être lancée.

Ce builtin évite à une méthode de devoir relancer explicitement une demande après un code de réponse quelconque.

Un code de réponse est une valeur entière représentant une condition d'erreur spécifique à une application.

NOTE Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

Structure lexicale

`retry_on_all_response_codes` (void)

L'élément lexical du builtin `retry_on_all_response_codes` est présenté dans le Tableau 85.

Tableau 85 – Builtin retry_on_all_response_codes

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

Exemple

```
retry_on_all_response_codes();
```

6.3.36 Builtin RETRY_ON_COMM_ERROR**Profils de communication: HART****Objet: Relancer si une erreur de communication est détectée**

Le builtin RETRY_ON_COMM_ERROR définit le masque de relance sur erreur de communication, de sorte que la commande actuelle soit relancée si une erreur de communication est détectée dans le dispositif ou l'application EDD lors de l'envoi de la commande. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

```
RETRY_ON_COMM_ERROR(VOID)
```

L'élément lexical du builtin RETRY_ON_COMM_ERROR est présenté dans le Tableau 86.

Tableau 86 – Builtin RETRY_ON_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.37 Builtin retry_on_comm_error**Profils de communication: FF, ISA100****Objet: Relancer si une erreur de communication est détectée**

Le builtin retry_on_comm_error commande la relance de la dernière demande effectuée par une méthode après une erreur de communication spécifiée. Une demande est relancée trois fois avant qu'elle n'échoue et qu'un retour d'erreur ne soit retourné.

Ce builtin évite à une méthode de devoir relancer explicitement une demande après réception d'une erreur de communication spécifiée.

Les codes d'erreur de communication sont définis par les consortiums. Leurs chaînes associées peuvent être définies dans un dictionnaire de texte. Une erreur de communication égale à zéro n'est pas assimilée à une erreur de communication.

Structure lexicale

```
retry_on_comm_error(error)
```

Les éléments lexicaux du builtin `retry_on_comm_error` sont présentés dans le Tableau 87.

Tableau 87 – Builtin `retry_on_comm_error`

Nom du paramètre	Type	Direction	Utilisation	Description
error	unsigned long	I	m	Erreurs de communication.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = retry_on_comm_error(E_ACCESS_HARDWARE_FAULT);
```

6.3.38 Builtin `RETRY_ON_COMM_STATUS`

Profils de communication: HART

Objet: Définir le masque de relance de statut de communication

Le builtin `RETRY_ON_COMM_STATUS` définit le ou les bits corrects dans le masque de relance de statut de communication, de sorte que tout bit défini correspondant du statut de communication déclenche la relance de la commande actuelle. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 1.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_method` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins `send`, `send_trans`, `send_command`, `send_command_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

```
RETRY_ON_COMM_STATUS(comm_status)
```

Les éléments lexicaux du builtin `RETRY_ON_COMM_STATUS` sont présentés dans le Tableau 88.

Tableau 88 – Builtin `RETRY_ON_COMM_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
comm_status	int	I	m	Nouveau masque de relance de statut de communication.
<return>	VOID	—	—	—

6.3.39 Builtin RETRY_ON_DEVICE_STATUS

Profils de communication: HART

Objet: Définir le masque de relance de statut du dispositif

Le builtin RETRY_ON_DEVICE_STATUS définit le ou les bits corrects dans le masque de relance de statut du dispositif, de sorte que tout bit défini correspondant du statut du dispositif déclenche la relance de la commande actuelle. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le premier octet retourné indique un statut de code de réponse lorsque le bit de poids fort (bit 7) a la valeur 0. Le deuxième octet retourné, statut du dispositif, est significatif uniquement si le premier retourne un statut de code de réponse. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction. La valeur de device_status peut être définie de manière à conduire le système à relancer la commande HART actuelle sur toute combinaison de bits de statut du dispositif. La valeur de device_status peut être définie sur toute valeur de 0x00 à 0x7F, où chaque bit correspond à un statut de dispositif retourné par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

RETRY_ON_DEVICE_STATUS(device_status)

Les éléments lexicaux du builtin RETRY_ON_DEVICE_STATUS sont présentés dans le Tableau 89.

Tableau 89 – Builtin RETRY_ON_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
device_status	int	l	m	Nouveau masque de relance de statut de communication.
<return>	VOID	—	—	—

6.3.40 Builtin RETRY_ON_NO_DEVICE (déconseillé)

Profils de communication: CS, GPE, HART

Objet: Relancer si aucun dispositif n'est détecté

Le builtin RETRY_ON_NO_DEVICE définit le masque de relance en l'absence de dispositif, de sorte que la commande actuelle soit relancée si aucun dispositif n'est détecté lors de l'envoi d'une commande.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

RETRY_ON_NO_DEVICE (VOID)

L'élément lexical du builtin RETRY_ON_NO_DEVICE est présenté dans le Tableau 90.

Tableau 90 – Builtin RETRY_ON_NO_DEVICE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.41 Builtin RETRY_ON_RESPONSE_CODE

Profils de communication: HART

Objet: Définir le masque de relance de code de réponse

Le builtin RETRY_ON_RESPONSE_CODE définit le bit correct dans le masque de relance de code de réponse, de sorte que la valeur de code de réponse spécifiée déclenche la relance de la commande actuelle. Les deux premiers octets de chaque réponse esclave contiennent des informations de statut. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0. La valeur de response_code peut être définie de manière à conduire le système à relancer la commande HART actuelle sur tout code de réponse. La valeur de response_code peut être définie sur toute valeur de 0 à 127.

Pour conduire à la relance de la commande HART actuelle sur plusieurs codes de réponse différents, ce builtin doit être appelé une fois pour chaque code de réponse à relancer.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les builtins send, send_trans, send_command, send_command_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

RETRY_ON_RESPONSE_CODE (response_code)

Les éléments lexicaux du builtin RETRY_ON_RESPONSE_CODE sont présentés dans le Tableau 91.

Tableau 91 – Builtin RETRY_ON_RESPONSE_CODE

Nom du paramètre	Type	Direction	Utilisation	Description
response_code	int	l	m	Code de réponse qu'il convient d'ajouter au masque de relance.
<return>	VOID	—	—	—

6.3.42 Builtin `retry_on_response_code`

Profils de communication: FF, ISA100

Objet: Définir le masque de relance de code de réponse

Le builtin `retry_on_response_code` commande la relance de la dernière demande effectuée par une méthode trois fois après réception d'un code de réponse spécifique.

Ce builtin évite à une méthode de devoir relancer explicitement une demande après réception d'un code de réponse spécifié.

Un code de réponse est une valeur représentant une condition d'erreur spécifique à une application.

NOTE Une valeur de code de réponse égale à zéro n'est pas assimilée à un code de réponse d'erreur. Chaque code de réponse et type de code de réponse est défini dans l'EDD.

Ce builtin relance la dernière demande de la méthode après réception du code de réponse spécifié.

Structure lexicale

```
retry_on_response_code (code)
```

Les éléments lexicaux du builtin `retry_on_response_code` sont présentés dans le Tableau 92.

Tableau 92 – Builtin `retry_on_response_code`

Nom du paramètre	Type	Direction	Utilisation	Description
code	unsigned long	I	m	Code de réponse en cours de résolution.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = retry_on_response_code(RC_INVALID_ENTRY);
```

6.3.43 Builtin `send`

Profils de communication: HART

Objet: Envoyer la commande spécifiée

Le builtin `send` envoie la commande spécifiée et retourne le code de réponse, le statut de communication et le statut de commande respectivement dans la matrice `cmd_status`. L'octet de statut du dispositif de la réponse du dispositif de terrain doit être examiné. Tout bit ayant une classe de statut "MORE" définie (1) dans le statut du dispositif doit déclencher l'émission automatique de la commande 48.

Les octets de la matrice `cmd_status` et les octets de statut de la commande 48 (si elle a été envoyée) sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. S'il existe une correspondance, la commande est relancée (la commande spécifiée ou la commande 48). Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Lorsqu'une commande 48 est envoyée indirectement en tant que résultat de réponse de statut du dispositif, les valeurs du masque xxx de la commande 48 définies à l'aide des builtins `XMTR_xxx` sont utilisées pour contrôler les interruptions et relances pour la commande 48. Les autres valeurs de masque et builtins associés sont utilisés pour contrôler les interruptions et relances de la commande directement envoyée (y compris la commande 48 si elle a été directement envoyée).

Les octets de statut pour la commande sont retournés dans le paramètre `cmd_status`, un pointeur vers une matrice de 3 octets de caractère non signé, qui est alloué dans la méthode appelante.

Structure lexicale

```
send(cmd_number, cmd_status)
```

Les éléments lexicaux du builtin `send` sont présentés dans le Tableau 93.

Tableau 93 – Builtin `send`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la commande à envoyer.
<code>cmd_status</code>	char*	O	m	Octets de statut de la commande d'envoi.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. <code>BI_SUCCESS</code> si la commande a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

Exemple

```
int result;
char cmd_status[STATUS_SIZE];
result = send(40, cmd_status);
```

Dans l'exemple ci-dessus, si l'un des bits de la classe de statut "MORE" définis dans `cmd_status[STATUS_DEVICE_STATUS]` est défini, la commande 48 est automatiquement envoyée avant le retour de la fonction d'envoi.

6.3.44 Builtin `send_all_values`

Profils de communication: FF, ISA100

Objet: Envoyer les valeurs modifiées au dispositif

Le builtin `send_all_values` envoie toutes les valeurs de la table des variables que la méthode a modifiées, mais qui n'ont pas été précédemment envoyées au dispositif. Cette action modifie les valeurs du dispositif de manière à correspondre aux valeurs définies par la méthode.

Ce builtin n'effectue aucune action pour une application EDD non en mémoire cache, car elle n'a pas de table de variables.

Structure lexicale

`send_all_values(VOID)`

L'élément lexical du builtin `send_all_values` est présenté dans le Tableau 94.

Tableau 94 – Builtin `send_all_values`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = send_all_values();
```

6.3.45 Builtin `send_command`

Profils de communication: HART

Objet: Envoyer la commande spécifiée

Le builtin `send_command` envoie la commande spécifiée. Les octets de statut/données de la commande sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. En cas de correspondance, la commande est relancée. Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Aucun octet de statut ou de données n'est retourné avec cette fonction. Les valeurs du masque xxx de la commande 48 définies à l'aide des builtins `XMTR_xxx` n'affectent pas les interruptions ou les relances pour le builtin `send_command`.

Structure lexicale

`send_command(cmd_number)`

Les éléments lexicaux du builtin `send_command` sont présentés dans le Tableau 95.

Tableau 95 – Builtin `send_command`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la commande à envoyer.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. <code>BI_SUCCESS</code> si la fonction a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

6.3.46 Builtin `send_command_trans`

Profils de communication: HART

Objet: Envoyer la commande avec la transaction spécifiée

Le builtin `send_command_trans` envoie la commande avec la transaction spécifiée au dispositif. Cette fonction doit être utilisée pour envoyer des commandes qui ont été définies avec des transactions multiples.

Le bultin `send_command_trans` envoie la commande spécifiée. Les octets de statut/données de la commande sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. En cas de correspondance, la commande est relancée. Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Aucun octet de statut ou de données n'est retourné avec cette commande. Les valeurs du masque xxx de la commande 48 définies à l'aide des bultins `XMTR_xxx` n'affectent pas les interruptions ou les relances pour le bultin `send_command`.

Structure lexicale

`send_command_trans(cmd_number, transaction)`

Les éléments lexicaux du bultin `send_command_trans` sont présentés dans le Tableau 96.

Tableau 96 – Bultin `send_command_trans`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la commande à envoyer.
<code>transaction</code>	int	I	m	Numéro de la TRANSACTION.
<code><return></code>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. <code>BI_SUCCESS</code> si la fonction a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

6.3.47 Bultin `send_trans`

Profils de communication: HART

Objet: Envoyer la commande avec la transaction spécifiée

Le bultin `send_trans` envoie une commande avec la transaction spécifiée au dispositif. Cette fonction doit être utilisée pour envoyer des commandes qui ont été définies avec des transactions multiples.

Le bultin `send_trans` envoie la commande spécifiée et retourne le code de réponse, le statut de communication et le statut du dispositif dans la matrice `cmd_status`. L'octet de statut du dispositif de la réponse du dispositif de terrain doit être examiné. Tout bit ayant une classe de statut "MORE" définie (1) dans le statut du dispositif doit déclencher l'émission automatique de la commande 48.

Les octets de la matrice `cmd_status` et les octets de statut de la commande 48 (si elle a été envoyée) sont ensuite soumis à un AND logique avec les masques d'interruption appropriés pour déterminer s'il convient d'interrompre la méthode. En cas de correspondance, la méthode est interrompue et les méthodes d'interruption (le cas échéant) sont exécutées. S'il n'existe pas de correspondance pour les masques d'interruption, les mêmes données sont soumises à un AND avec les masques de relance appropriés. S'il existe une correspondance, la commande est relancée (la commande spécifiée ou la commande 48). Cette procédure est répétée jusqu'à ce que la commande aboutisse, qu'une interruption se produise ou que le nombre maximal de relances soit atteint. Lorsqu'une commande 48 est envoyée indirectement en tant que résultat de réponse de statut du dispositif, les valeurs du masque xxx de la commande 48 définies à l'aide des bultins `XMTR_xxx` sont utilisées pour contrôler les interruptions et relances pour la commande 48. Les autres valeurs de masque et bultins associés sont utilisés pour contrôler les interruptions et relances de la commande directement envoyée (y compris la commande 48 si elle a été directement envoyée).

Les octets de statut pour la commande sont retournés dans le paramètre `cmd_status`, une matrice de 3 octets, alloué dans la méthode appelante.

Structure lexicale

```
send_trans(cmd_number, transaction, cmd_status)
```

Les éléments lexicaux du builtin `send_trans` sont présentés dans le Tableau 97.

Tableau 97 – Builtin `send_trans`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>cmd_number</code>	int	I	m	Numéro de la commande à envoyer.
<code>transaction</code>	int	I	m	Numéro de la TRANSACTION.
<code>cmd_status</code>	char*	O	m	Octets de statut de la commande d'envoi.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. <code>BI_SUCCESS</code> si la commande a abouti, <code>BI_COMM_ERROR</code> si une erreur s'est produite lors de l'envoi de la commande ou <code>BI_NO_DEVICE</code> si aucun dispositif n'a été détecté.

6.3.48 Builtin `send_value`

Profils de communication: FF, ISA100

Objet: Envoyer une valeur de table de variables au dispositif

Le builtin `send_value` commande l'envoi de la valeur de la variable spécifiée dans la mémoire cache au dispositif, de manière à mettre à jour la valeur du dispositif. Ce builtin n'effectue aucune action dans une application EDD non en mémoire cache, car une application EDD non en mémoire cache ne comporte pas de mémoire cache.

Certaines valeurs sont `READ_ONLY` dans le dispositif et ne peuvent pas être écrites. Toute tentative d'une telle écriture provoque le retour d'une erreur de builtin.

Les paramètres `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la variable en question.

Structure lexicale

```
send_value(id, member_id)
```

Les éléments lexicaux du builtin `send_value` sont présentés dans le Tableau 98.

Tableau 98 – Builtin `send_value`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>id</code>	unsigned long	I	m	ID d'élément de la VARIABLE à lire.
<code>member_id</code>	unsigned long	I	m	ID de membre de la VARIABLE à lire.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long id;
id = ITEM_ID(PARAM.HI_LIM);
status = edit_device_value ("|en|Enter new HI_LIM value:\n", 0, 0, 0, id,
0); status = send_value(id, 0);
```

6.3.49 Builtin send_value2**Profils de communication: FF, ISA100**

Objet: Envoyer la valeur d'une variable, d'un membre d'enregistrement ou d'un élément de matrice au dispositif

Le builtin `send_value2` commande l'envoi de la valeur de la variable, du membre d'enregistrement ou de l'élément de matrice spécifié dans la mémoire cache au dispositif, de manière à mettre à jour la valeur du dispositif. Ce builtin n'effectue aucune action dans une application EDD non en mémoire cache, car une application EDD non en mémoire cache ne comporte pas de mémoire cache.

Certaines valeurs sont `READ_ONLY` dans le dispositif et ne peuvent pas être écrites. Toute tentative d'une telle écriture provoque le retour d'une erreur de builtin.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une variable simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Si le paramètre `id` référence un enregistrement et que le paramètre `member_id` a la valeur zéro, l'ensemble de l'enregistrement est écrit dans le dispositif.

De même, si le paramètre `id` référence une matrice et que le paramètre `member_id` a la valeur zéro, l'ensemble de la matrice est écrit dans le dispositif.

Structure lexicale

```
send_value2(blk_id, blk_num, id, member_id)
```

Les éléments lexicaux du builtin `send_value2` sont présentés dans le Tableau 99.

Tableau 99 – Builtin send_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	m	ID d'élément de la VARIABLE à lire.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à lire.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long aiBlock;
unsigned long id;
```

```
aiBlock = ITEM_ID(__analog_input_block);
id = MEMBER_ID(HI_LIM);
```

```
status = edit_device_value2("|en|Enter new HI_LIM value:\n", 0, 0, 0, 0, 0,
aiBlock, 0, id, 0);
status = send_value2(aiBlock, 0, id, 0);
```

6.3.50 Builtin SET_NUMBER_OF_RETRIES**Profils de communication: CS, GPE, HART****Objet: Définir le nombre de relances d'une commande**

Le builtin SET_NUMBER_OF_RETRIES spécifie le nombre de relances d'une commande en fonction des masques de relance appropriés. Cette valeur est définie par défaut sur 0 au début de chaque méthode.

Structure lexicale

```
SET_NUMBER_OF_RETRIES(n)
```

Les éléments lexicaux du builtin SET_NUMBER_OF_RETRIES sont présentés dans le Tableau 100.

Tableau 100 – Builtin SET_NUMBER_OF_RETRIES

Nom du paramètre	Type	Direction	Utilisation	Description
n	int	I	m	Nombre de relances d'une commande.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377.

6.3.51 Builtin WriteCommand**Profils de communication: CS, GPE, PB, PN****Objet: Exécuter une commande d'écriture**

Le builtin WriteCommand permet d'écrire des variables avec une commande spécifiée.

Structure lexicale

WriteCommand(command_identifiant)

Les éléments lexicaux du builtin WriteCommand sont présentés dans le Tableau 101.

Tableau 101 – Builtin WriteCommand

Nom du paramètre	Type	Direction	Utilisation	Description
command_identifiant	reference	I	m	Identifiant de la COMMAND à exécuter.
<return>	long	O	m	Valeur d'erreur spécifique au protocole, dont le codage est défini par le protocole ou dispositif spécifique.

6.3.52 Builtin XMTR_ABORT_ON_ALL_COMM_STATUS

Profils de communication: HART

Objet: Définir l'interruption pour tous les statuts de communication de commande 48

Le builtin XMTR_ABORT_ON_ALL_COMM_STATUS définit tous les bits dans le masque d'interruption de statut de communication d'une commande 48. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de statut de communication d'une commande 48. Le statut de communication est défini comme étant le premier octet retourné dans une transaction, lorsque le bit 7 de cet octet est défini. Le statut de communication peut être défini sur toute valeur de 0x00 à 0x7F, où chaque bit correspond à une erreur de communication reçue par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_ALL_COMM_STATUS (VOID)

L'élément lexical du builtin XMTR_ABORT_ON_ALL_COMM_STATUS est présenté dans le Tableau 102.

Tableau 102 – Builtin XMTR_ABORT_ON_ALL_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.53 Builtin XMTR_ABORT_ON_ALL_DATA

Profils de communication: HART

Objet: Définir l'interruption pour toutes les données de commande 48

Le builtin XMTR_ABORT_ON_ALL_DATA définit tous les bits dans les masques d'interruption de données de la commande 48. Cela conduit le système à interrompre si un bit quelconque de la commande 48 est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les fonctions `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY` des builtins.

Structure lexicale

`XMTR_ABORT_ON_ALL_DATA(VOID)`

L'élément lexical du builtin `XMTR_ABORT_ON_ALL_DATA` est présenté dans le Tableau 103.

Tableau 103 – Builtin `XMTR_ABORT_ON_ALL_DATA`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.54 Builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS`

Profils de communication: HART

Objet: Définir l'interruption pour tous les statuts de communication de commande 48

Le builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS` définit tous les bits dans le masque d'interruption de statut du dispositif d'une commande 48. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de statut du dispositif d'une commande 48. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_ABORT_ON_ALL_DEVICE_STATUS(VOID)`

L'élément lexical du builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS` est présenté dans le Tableau 104.

Tableau 104 – Builtin `XMTR_ABORT_ON_ALL_DEVICE_STATUS`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.55 Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES

Profils de communication: HART

Objet: Définir l'interruption pour tous les codes de réponse de commande 48

Le builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES définit tous les bits dans le masque d'interruption de code de réponse d'une commande 48. Cela conduit le système à interrompre la méthode actuelle si le dispositif retourne une valeur de code de réponse d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_ALL_RESPONSE_CODES (VOID)

L'élément lexical du builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES est présenté dans le Tableau 105.

Tableau 105 – Builtin XMTR_ABORT_ON_ALL_RESPONSE_CODES

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.56 Builtin XMTR_ABORT_ON_COMM_ERROR

Profils de communication: HART

Objet: Interrompre si une erreur de communication est détectée pendant la commande 48

Le builtin XMTR_ABORT_ON_COMM_ERROR définit le masque d'interruption sur erreur de communication d'une commande 48, de sorte que la méthode soit interrompue si une erreur de communication est détectée lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_COMM_ERROR (VOID)

L'élément lexical du builtin XMTR_ABORT_ON_COMM_ERROR est présenté dans le Tableau 106.

Tableau 106 – Builtin XMTR_ABORT_ON_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.57 Builtin XMTR_ABORT_ON_COMM_STATUS**Profils de communication: HART****Objet: Ajouter un masque d'interruption de statut à la commande 48**

Le builtin XMTR_ABORT_ON_COMM_STATUS définit le ou les bits corrects dans le masque d'interruption de statut de communication d'une commande 48, de sorte que la valeur spécifiée de statut de communication d'une commande 48 conduise à l'interruption de la méthode. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 1. La valeur `comm_status_value` peut être définie sur toute valeur de 0 à 127, où chaque bit correspond à une erreur de communication reçue par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_ABORT_ON_COMM_STATUS(comm_status)`

Les éléments lexicaux du builtin XMTR_ABORT_ON_COMM_STATUS sont présentés dans le Tableau 107.

Tableau 107 – Builtin XMTR_ABORT_ON_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<code>comm_status</code>	<code>int</code>	<code>l</code>	<code>m</code>	Nouveau masque d'interruption de statut de communication.
<return>	VOID	—	—	—

6.3.58 Builtin XMTR_ABORT_ON_DATA**Profils de communication: HART****Objet: Interrompre si un bit de données spécifié dans la commande 48 est défini**

Le builtin XMTR_ABORT_ON_DATA définit le ou les bits corrects dans le masque d'interruption d'une commande 48, de sorte que la méthode soit interrompue si le bit spécifié est défini dans le champ de données retourné par la commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_DATA(data, bit)

Les éléments lexicaux du builtin XMTR_ABORT_ON_DATA sont présentés dans le Tableau 108.

Tableau 108 – Builtin XMTR_ABORT_ON_DATA

Nom du paramètre	Type	Direction	Utilisation	Description
data	int	l	m	Index base zéro dans le champ de données d'une commande 48.
bit	int	l	m	Numéro de bit (0 à 7). Le bit 0 est le bit de poids faible.
<return>	VOID	—	—	—

6.3.59 Builtin XMTR_ABORT_ON_DEVICE_STATUS

Profils de communication: HART

Objet: Ajouter un masque d'interruption de statut à la commande 48

Le builtin XMTR_ABORT_ON_DEVICE_STATUS définit le ou les bits corrects dans le masque d'interruption de statut du dispositif d'une commande 48, de sorte que tout bit défini correspondant de la commande 48 conduise à l'interruption de la méthode. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction. La valeur device_status peut être définie à toute valeur de 0 à 127, où chaque bit correspond à un statut du dispositif retourné par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les fonctions send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_DEVICE_STATUS(device_status)

Les éléments lexicaux du builtin XMTR_ABORT_ON_DEVICE_STATUS sont présentés dans le Tableau 109.

Tableau 109 – Builtin XMTR_ABORT_ON_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
device_status	int	l	m	Nouveau masque d'interruption de statut du dispositif.
<return>	VOID	—	—	—

6.3.60 Builtin XMTR_ABORT_ON_NO_DEVICE (déconseillé)

Profils de communication:

Objet: Interrompre si aucun dispositif n'est détecté pendant la commande 48

Le builtin XMTR_ABORT_ON_NO_DEVICE définit le masque d'absence de dispositif d'une commande 48, de sorte que la méthode soit interrompue si aucun dispositif n'est détecté lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_NO_DEVICE (VOID)

L'élément lexical du builtin XMTR_ABORT_ON_NO_DEVICE est présenté dans le Tableau 110.

Tableau 110 – Builtin XMTR_ABORT_ON_NO_DEVICE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.61 Builtin XMTR_ABORT_ON_RESPONSE_CODE

Profils de communication: HART

Objet: Ajouter un masque d'interruption de code de réponse à la commande 48

Le builtin XMTR_ABORT_ON_RESPONSE_CODE définit le bit correct dans le masque d'interruption de code de réponse d'une commande 48, de sorte que la valeur spécifiée du code de réponse d'une commande 48 conduise à l'interruption de la méthode. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_ABORT_ON_RESPONSE_CODE (response_code)

Les éléments lexicaux du builtin XMTR_ABORT_ON_RESPONSE_CODE sont présentés dans le Tableau 111.

Tableau 111 – Builtin XMTR_ABORT_ON_RESPONSE_CODE

Nom du paramètre	Type	Direction	Utilisation	Description
response_code	int	l	m	Nouveau masque d'interruption de code de réponse.
<return>	VOID	—	—	—

6.3.62 Builtin XMTR_IGNORE_ALL_COMM_STATUS**Profils de communication: HART****Objet: Ignorer tous les bits de statut de commande 48**

Le builtin XMTR_IGNORE_ALL_COMM_STATUS efface tous les bits dans le masque de relance de statut de communication et le masque d'interruption de statut de communication d'une commande 48. Cela conduit le système à ignorer tous les bits dans la valeur de statut de communication. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_IGNORE_ALL_COMM_STATUS (VOID)

L'élément lexical du builtin XMTR_IGNORE_ALL_COMM_STATUS est présenté dans le Tableau 112.

Tableau 112 – Builtin XMTR_IGNORE_ALL_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.63 Builtin XMTR_IGNORE_ALL_DATA**Profils de communication: HART****Objet: Ignorer tous les bits de statut du dispositif de commande 48**

Le builtin XMTR_IGNORE_ALL_DATA efface tous les bits dans le masque de relance de données et le masque d'interruption de données d'une commande 48. Cela conduit le système à ignorer toutes les données dans le message de réponse d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir send_command pour la mise en œuvre des masques. Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par cette fonction est utilisé par les fonctions send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_IGNORE_ALL_DATA (VOID)

L'élément lexical du builtin XMTR_IGNORE_ALL_DATA est présenté dans le Tableau 113.

Tableau 113 – Builtin XMTR_IGNORE_ALL_DATA

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.64 Builtin XMTR_IGNORE_ALL_DEVICE_STATUS

Profils de communication: HART

Objet: Ignorer tous les bits de statut du dispositif de commande 48

Le builtin XMTR_IGNORE_ALL_DEVICE_STATUS efface tous les bits dans le masque de relance de statut du dispositif et le masque d'interruption de statut du dispositif d'une commande 48. Cela conduit le système à ignorer tous les bits dans l'octet de statut du dispositif d'une commande 48. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_IGNORE_ALL_DEVICE_STATUS (VOID)

L'élément lexical du builtin XMTR_IGNORE_ALL_DEVICE_STATUS est présenté dans le Tableau 114.

Tableau 114 – Builtin XMTR_IGNORE_ALL_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.65 Builtin XMTR_IGNORE_ALL_RESPONSE_CODES

Profils de communication: HART

Objet: Ignorer tous les codes de réponse de commande 48

Le builtin XMTR_IGNORE_ALL_RESPONSE_CODES efface tous les bits dans le masque de relance de code de réponse et le masque d'interruption de code de réponse d'une commande 48. Cela conduit le système à ignorer toutes les valeurs de code de réponse retournées par le dispositif.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_IGNORE_ALL_RESPONSE_CODES(VOID)`

L'élément lexical du builtin `XMTR_IGNORE_ALL_RESPONSE_CODES` est présenté dans le Tableau 115.

Tableau 115 – Builtin XMTR_IGNORE_ALL_RESPONSE_CODES

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.66 Builtin XMTR_IGNORE_COMM_ERROR

Profils de communication: HART

Objet: Ignorer si une erreur de communication est détectée pendant la commande 48

Le builtin `XMTR_IGNORE_COMM_ERROR` efface tous les bits dans le masque de relance sur erreur de communication et le masque d'interruption sur erreur de communication d'une commande 48, de sorte qu'une condition d'erreur de communication soit ignorée lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_IGNORE_COMM_ERROR(VOID)`

L'élément lexical du builtin `XMTR_IGNORE_COMM_ERROR` est présenté dans le Tableau 116.

Tableau 116 – Builtin XMTR_IGNORE_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.67 Builtin XMTR_IGNORE_COMM_STATUS

Profils de communication: HART

Objet: Ignorer des bits de statut de communication de commande 48 spécifiques

Le builtin XMTR_IGNORE_COMM_STATUS efface le ou les bits corrects dans le masque de relance de statut de communication et le masque d'interruption de statut de communication d'une commande 48, de sorte que les bits spécifiés dans la valeur du statut de communication d'une commande 48 soient ignorés. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_IGNORE_COMM_STATUS(comm_status)

Les éléments lexicaux du builtin XMTR_IGNORE_COMM_STATUS sont présentés dans le Tableau 117.

Tableau 117 – Builtin XMTR_IGNORE_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
comm_status	int	l	m	Nouveau masque d'interruption et de relance de statut de communication.
<return>	VOID	—	—	—

6.3.68 Builtin XMTR_IGNORE_DATA

Profils de communication: HART

Objet: Ignorer des bits de données de commande 48 spécifiques

Le builtin XMTR_IGNORE_DATA doit réinitialiser le bit spécifié, dans l'octet spécifié, dans le masque de relance des données et le masque d'interruption des données de la commande 48, de sorte que la méthode ignore le bit spécifié dans le champ de données retourné par la commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir send_command pour la mise en œuvre des masques.

Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les fonctions send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display and DISPLAY des builtins.

Structure lexicale

XMTR_IGNORE_DATA(data, bit)

Les éléments lexicaux du builtin XMTR_IGNORE_DATA sont présentés dans le Tableau 118.

Tableau 118 – Builtin XMTR_IGNORE_DATA

Nom du paramètre	Type	Direction	Utilisation	Description
Data	int	I	m	Index base zéro dans le champ de données d'une commande 48.
Bit	int	I	m	Numéro de bit (0 à 7). Le bit 0 est le bit de poids faible.
<return>	VOID	—	—	—

6.3.69 Builtin XMTR_IGNORE_DEVICE_STATUS

Profils de communication: HART

Objet: Ignorer des bits de statut de commande 48 spécifiques

Le builtin XMTR_IGNORE_DEVICE_STATUS efface le ou les bits corrects dans le masque de relance de statut du dispositif et le masque d'interruption de statut du dispositif d'une commande 48, de sorte que les bits spécifiés dans l'octet de statut du dispositif d'une commande 48 soient ignorés. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_IGNORE_DEVICE_STATUS(device_status)

Les éléments lexicaux du builtin XMTR_IGNORE_DEVICE_STATUS sont présentés dans le Tableau 119.

Tableau 119 – Builtin XMTR_IGNORE_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
device_status	int	I	m	Bits à effacer dans le masque d'interruption et de relance.
<return>	VOID	—	—	—

6.3.70 Builtin XMTR_IGNORE_NO_DEVICE (déconseillé)

Profils de communication: HART

Objet: Ignorer si aucun dispositif n'est détecté pendant la commande 48

Le builtin XMTR_IGNORE_NO_DEVICE définit le masque de relance en l'absence de dispositif et le masque d'interruption en l'absence de dispositif d'une commande 48 pour indiquer qu'il convient d'ignorer la condition d'absence de dispositif lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_IGNORE_NO_DEVICE (VOID)`

L'élément lexical du builtin `XMTR_IGNORE_NO_DEVICE` est présenté dans le Tableau 120.

Tableau 120 – Builtin `XMTR_IGNORE_NO_DEVICE`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.71 Builtin `XMTR_IGNORE_RESPONSE_CODE`

Profils de communication: HART

Objet: Ignorer des codes de réponse de commande 48 spécifiques

Le builtin `XMTR_IGNORE_RESPONSE_CODE` efface le bit correct dans le masque de relance de code de réponse et le masque d'interruption de code de réponse d'une commande 48, de sorte que la valeur spécifiée de code de réponse d'une commande 48 soit ignorée. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_IGNORE_RESPONSE_CODE (response_code)`

Les éléments lexicaux du builtin `XMTR_IGNORE_RESPONSE_CODE` sont présentés dans le Tableau 121.

Tableau 121 – Builtin `XMTR_IGNORE_RESPONSE_CODE`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>response_code</code>	int	l	m	Nouveau masque de code de réponse.
<return>	VOID	—	—	—

6.3.72 Builtin XMTR_RETRY_ON_ALL_COMM_STATUS

Profils de communication: HART

Objet: Définir la relance pour tous les statuts de communication de commande 48

Le builtin XMTR_RETRY_ON_ALL_COMM_STATUS définit tous les bits dans le masque de relance de statut de communication de la commande 48. Cela conduit le système à relancer la commande 48 si le dispositif retourne une valeur de statut de communication d'une commande 48. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction HART, lorsque le bit 7 de cet octet est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les fonctions `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY` des builtins.

Structure lexicale

`XMTR_RETRY_ON_ALL_COMM_STATUS(VOID)`

L'élément lexical du builtin XMTR_RETRY_ON_ALL_COMM_STATUS est présenté dans le Tableau 122.

Tableau 122 – Builtin XMTR_RETRY_ON_ALL_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.73 Builtin XMTR_RETRY_ON_ALL_DATA

Profils de communication: HART

Objet: Définir la relance pour tous les statuts du dispositif de commande 48

Le builtin XMTR_RETRY_ON_ALL_DATA définit tous les bits dans le masque de relance de données d'une commande 48. Cela conduit le système à relancer si un bit quelconque dans la commande 48 est défini.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir `send_command` pour la mise en œuvre des masques. Voir `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les fonctions `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` and `DISPLAY` des builtins.

Structure lexicale

`XMTR_RETRY_ON_ALL_DATA(VOID)`

L'élément lexical du builtin XMTR_RETRY_ON_ALL_DATA est présenté dans le Tableau 123.

Tableau 123 – Builtin XMTR_RETRY_ON_ALL_DATA

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.74 Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS**Profils de communication: HART****Objet: Définir la relance pour tous les statuts du dispositif de commande 48**

Le builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS définit tous les bits dans le masque de relance de statut du dispositif d'une commande 48. Cela conduit le système à relancer la commande 48 si le dispositif retourne une valeur de statut du dispositif d'une commande 48. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_ALL_DEVICE_STATUS (VOID)

L'élément lexical du builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS est présenté dans le Tableau 124.

Tableau 124 – Builtin XMTR_RETRY_ON_ALL_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.75 Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES**Profils de communication: HART****Objet: Définir la relance pour tous les codes de réponse de commande 48**

Le builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES définit tous les bits dans le masque de relance de code de réponse d'une commande 48. Cela conduit le système à relancer la commande 48 du système si le dispositif retourne une valeur de code de réponse d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_ALL_RESPONSE_CODES (VOID)

L'élément lexical du builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES est présenté dans le Tableau 125.

Tableau 125 – Builtin XMTR_RETRY_ON_ALL_RESPONSE_CODES

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.76 Builtin XMTR_RETRY_ON_COMM_ERROR

Profils de communication: HART

Objet: Relancer si une erreur de communication est détectée pendant la commande 48

Le builtin XMTR_RETRY_ON_COMM_ERROR définit le masque de relance sur erreur de communication d'une commande 48, de sorte que la commande 48 soit relancée si une erreur de communication est détectée lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_COMM_ERROR (VOID)

L'élément lexical du builtin XMTR_RETRY_ON_COMM_ERROR est présenté dans le Tableau 126.

Tableau 126 – Builtin XMTR_RETRY_ON_COMM_ERROR

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.77 Builtin XMTR_RETRY_ON_COMM_STATUS

Profils de communication: HART

Objet: Ajouter un masque de relance de statut à la commande 48

Le builtin XMTR_RETRY_ON_COMM_STATUS définit le ou les bits corrects dans le masque de relance de statut de communication d'une commande 48, de sorte que la valeur spécifiée de statut de communication d'une commande 48 conduise à la relance de la commande 48. Le statut de communication est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 1.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_RETRY_ON_COMM_STATUS(comm_status)`

Les éléments lexicaux du builtin `XMTR_RETRY_ON_COMM_STATUS` sont présentés dans le Tableau 127.

Tableau 127 – Builtin XMTR_RETRY_ON_COMM_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
<code>comm_status</code>	<code>int</code>	<code>l</code>	<code>m</code>	Nouveau masque de relance de statut de communication.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.78 Builtin XMTR_RETRY_ON_DATA

Profils de communication: HART

Objet: Relancer si un bit de données spécifié dans la commande 48 est défini

Le builtin `XMTR_RETRY_ON_DATA` définit le ou les bits spécifiés dans le masque de relance de données d'une commande 48, de sorte que la commande 48 soit relancée si le bit spécifié est défini dans le champ de données retourné par la commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction `send_command` pour la mise en œuvre des masques. Voir le builtin `ABORT_ON_RESPONSE_CODE` pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins `send`, `send_trans`, `ext_send_command`, `ext_send_command_trans`, `get_more_status`, `display` et `DISPLAY`.

Structure lexicale

`XMTR_RETRY_ON_DATA(data, bit)`

Les éléments lexicaux du builtin `XMTR_RETRY_ON_DATA` sont présentés dans le Tableau 128.

Tableau 128 – Builtin XMTR_RETRY_ON_DATA

Nom du paramètre	Type	Direction	Utilisation	Description
<code>data</code>	<code>int</code>	<code>l</code>	<code>m</code>	Index base zéro dans le champ de données d'une commande 48.
<code>bit</code>	<code>int</code>	<code>l</code>	<code>m</code>	Numéro de bit (0 à 7). Le bit 0 est le bit de poids faible.
<code><return></code>	<code>VOID</code>	—	—	—

6.3.79 Builtin XMTR_RETRY_ON_DEVICE_STATUS

Profils de communication: HART

Objet: Ajouter un masque de relance de statut à la commande 48

Le builtin XMTR_RETRY_ON_DEVICE_STATUS définit le ou les bits corrects dans le masque de relance de statut du dispositif d'une commande 48, de sorte que tout bit défini correspondant du statut du dispositif d'une commande 48 conduise à la relance de la commande 48. Le statut du dispositif est défini comme étant le deuxième octet de données retourné dans une transaction.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_DEVICE_STATUS(device_status)

Les éléments lexicaux du builtin XMTR_RETRY_ON_DEVICE_STATUS sont présentés dans le Tableau 129.

Tableau 129 – Builtin XMTR_RETRY_ON_DEVICE_STATUS

Nom du paramètre	Type	Direction	Utilisation	Description
device_status	INT4	I	m	Nouveau masque de relance de statut du dispositif.
<return>	VOID	—	—	—

6.3.80 Builtin XMTR_RETRY_ON_NO_DEVICE (déconseillé)

Profils de communication: HART

Objet: Relancer si aucun dispositif n'est détecté pendant la commande 48

Le builtin XMTR_RETRY_ON_NO_DEVICE définit le masque de relance en l'absence de dispositif d'une commande 48, de sorte que la commande 48 soit relancée si aucun dispositif n'est détecté lors de l'envoi d'une commande 48.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_NO_DEVICE(VOID)

L'élément lexical du builtin XMTR_RETRY_ON_NO_DEVICE est présenté dans le Tableau 130.

Tableau 130 – Builtin XMTR_RETRY_ON_NO_DEVICE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.3.81 Builtin XMTR_RETRY_ON_RESPONSE_CODE**Profils de communication: HART****Objet: Ajouter un masque de relance de code de réponse à la commande 48**

Le builtin XMTR_RETRY_ON_RESPONSE_CODE définit le bit correct dans le masque de relance de code de réponse d'une commande 48, de sorte que la valeur de code de réponse spécifiée d'une commande 48 conduise à la relance de la commande 48. Le code de réponse est défini comme étant le premier octet de données retourné dans une transaction, lorsque le bit 7 de cet octet est 0.

Les masques de relance et d'interruption sont réinitialisés à leurs valeurs par défaut au début de chaque méthode, de sorte que la nouvelle valeur du masque est valide uniquement pendant la méthode actuelle. Voir la fonction send_command pour la mise en œuvre des masques. Voir le builtin ABORT_ON_RESPONSE_CODE pour obtenir une liste des masques disponibles et leurs valeurs par défaut.

Le masque affecté par ce builtin est utilisé par les builtins send, send_trans, ext_send_command, ext_send_command_trans, get_more_status, display et DISPLAY.

Structure lexicale

XMTR_RETRY_ON_RESPONSE_CODE(response_code)

L'élément lexical du builtin XMTR_RETRY_ON_RESPONSE_CODE est présenté dans le Tableau 131.

Tableau 131 – Builtin XMTR_RETRY_ON_RESPONSE_CODE

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	VOID	—	—	—

6.4 Builtins d'action**6.4.1 Builtin fgetval****Profils de communication: CS, GPE, HART, PB, PN****Objet: Obtenir une valeur de VARIABLE**

Le builtin fgetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

fgetval(VOID)

L'élément lexical du builtin fgetval est présenté dans le Tableau 132.

Tableau 132 – Builtin fgetval

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	double	O	m	Valeur de la VARIABLE.

Exemple

```
double fval;
fval = fgetval();
fval = fval + 10;
```

6.4.2 Builtin fsetval

Profils de communication: CS, GPE, HART, PB, PN

Objet: Définir une valeur de VARIABLE

Le builtin fsetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
fsetval(value)
```

Les éléments lexicaux du builtin fsetval sont présentés dans le Tableau 133.

Tableau 133 – Builtin fsetval

Nom du paramètre	Type	Direction	Utilisation	Description
value	double	I	m	Nouvelle valeur pour la VARIABLE.
<return>	double	O	m	Valeur de la VARIABLE.

6.4.3 Builtin get_date

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE de date

Le builtin get_date est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions préédition, postédition, postlecture ou préécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue ou écrite par une application, une méthode utilisateur ou une méthode d'édition.

Structure lexicale

```
get_date(data, size)
```

Les éléments lexicaux du builtin `get_date` sont présentés dans le Tableau 134.

Tableau 134 – Builtin `get_date`

Nom du paramètre	Type	Direction	Utilisation	Description
data	char[]	O	m	Valeur retournée de la VARIABLE. La structure de cette mémoire tampon d'octet est définie à l'Annexe B de l'IEC 61804-3: __6.
size	long	I/O	m	Longueur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status, date_size;
char date_value[8];

date_size = 8;
status = get_date(date_value, &date_size);
```

6.4.4 Builtin `get_double`

Profils de communication: FF, ISA100

Objet: Obtenir d'une VARIABLE une valeur double précision

Le builtin `get_double` est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur de la VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions prédiction, postédition, postlecture ou préécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue par une application ou une autre méthode.

Structure lexicale

```
get_double(value)
```

Les éléments lexicaux du builtin `get_double` sont présentés dans le Tableau 135.

Tableau 135 – Builtin `get_double`

Nom du paramètre	Type	Direction	Utilisation	Description
value	double	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
```

⁶ En cours d'élaboration. Stade au moment de la publication: IEC RPUB 61804-3:2020.

```
double value;
status = get_double(&value);
```

6.4.5 Builtin get_float

Profils de communication: FF, ISA100

Objet: Obtenir d'une VARIABLE une valeur à virgule flottante

Le builtin get_float est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par un dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions préédition, postédition, postlecture ou préécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue par une application, une méthode utilisateur ou une méthode d'édition.

Structure lexicale

```
get_float(value)
```

Les éléments lexicaux du builtin get_float sont présentés dans le Tableau 136.

Tableau 136 – Builtin get_float

Nom du paramètre	Type	Direction	Utilisation	Description
value	float	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
status = get_float(&value);
```

6.4.6 Builtin get_signed

Profils de communication: FF, ISA100

Objet: Obtenir la valeur signée mise à l'échelle d'une VARIABLE

Le builtin get_signed est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions préédition, postédition, postlecture ou préécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue par une application, une méthode utilisateur ou une méthode d'édition.

Structure lexicale

```
get_signed(value)
```

Les éléments lexicaux du builtin get_signed sont présentés dans le Tableau 137.

Tableau 137 – Builtin get_signed

Nom du paramètre	Type	Direction	Utilisation	Description
value	long	O	m	Valeur retournée de la VARIABLE signée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long value;
status = get_signed(&value);
```

6.4.7 Builtin get_string

Profils de communication: FF, ISA100

Objet: Obtenir la valeur de chaîne d'une VARIABLE

Le builtin get_string est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par un dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit sauvegardée dans l'application EDD.

Les VARIABLES de ces types de données doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions préédition, postédition, postlecture ou postécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue par une application, une méthode utilisateur ou une méthode d'édition.

Structure lexicale

```
get_string(string, len)
```

Les éléments lexicaux du builtin get_string sont présentés dans le Tableau 138.

Tableau 138 – Builtin get_string

Nom du paramètre	Type	Direction	Utilisation	Description
string	char*	O	m	Mémoire tampon pour la chaîne retournée.
len	long*	I/O	m	Taille maximale de la chaîne. Longueur retournée de la chaîne.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long len;
char buf[256];
status = get_string(buf, &len);
```

6.4.8 Builtin get_unsigned**Profils de communication: FF, ISA100****Objet: Obtenir d'une VARIABLE une valeur non signée**

Le builtin `get_unsigned` est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Les méthodes associées aux actions préédition, postédition, postlecture ou préécriture sont exécutées lorsqu'une VARIABLE pour laquelle une de ces actions est définie en tant qu'attribut est lue par une application, une méthode utilisateur ou une méthode d'édition.

Structure lexicale

```
get_unsigned(value)
```

Les éléments lexicaux du builtin `get_unsigned` sont présentés dans le Tableau 139.

Tableau 139 – Builtin get_unsigned

Nom du paramètre	Type	Direction	Utilisation	Description
value	unsigned long	O	m	Valeur retournée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
status = get_unsigned(&value);
```

6.4.9 Builtin igetval**Profils de communication: CS, GPE, HART, PB, PN****Objet: Obtenir la valeur d'une VARIABLE**

Le builtin `igetval` est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

`igetval (VOID)`

L'élément lexical du builtin `igetval` est présenté dans le Tableau 140.

Tableau 140 – Builtin `igetval`

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	long long	O	m	Valeur de la VARIABLE.

6.4.10 Builtin `isetval`

Profils de communication: CS, GPE, HART, PB, PN

Objet: Définir une valeur de VARIABLE

Le builtin `isetval` est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

`isetval (value)`

Les éléments lexicaux du builtin `isetval` sont présentés dans le Tableau 141.

Tableau 141 – Builtin `isetval`

Nom du paramètre	Type	Direction	Utilisation	Description
value	long long	I	m	Nouvelle valeur de la VARIABLE.
<return>	long long	O	m	Valeur de la VARIABLE.

6.4.11 Builtin `isOffline`

Profils de communication: CS, FF, GPE, HART, ISA100, PB, PN

Objet: Déterminer si l'application EDD peut actuellement communiquer avec le dispositif de terrain

Le builtin `isOffline` retourne une valeur différente de zéro, lorsque l'état actuel de l'application EDD (dans le contexte de la méthode appelante) est "Offline".

Structure lexicale

`isOffline()`

L'élément lexical du builtin `isOffline` est présenté dans le Tableau 142.

Tableau 142 – Builtin isOffline

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	int	O	m	Valeur différente de zéro si l'application EDD est hors ligne et ne peut pas communiquer avec le dispositif, 0 si l'application EDD peut communiquer avec le dispositif.

Exemple

```
int offline;
offline = isOffline();
```

6.4.12 Builtin lgetval

Profils de communication: CS, GPE, PB, PN

Objet

Le builtin lgetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
lgetval (VOID)
```

L'élément lexical du builtin lgetval est présenté dans le Tableau 143.

Tableau 143 – Builtin lgetval

Nom du paramètre	Type	Direction	Utilisation	Description
<return>	int	O	m	Valeur de la VARIABLE.

6.4.13 Builtin lsetval

Profils de communication: CS, GPE, PB, PN

Objet: Définir une valeur longue

Le builtin lsetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
lsetval (value)
```

Les éléments lexicaux du builtin lsetval sont présentés dans le Tableau 144.

Tableau 144 – Builtin lsetval

Nom du paramètre	Type	Direction	Utilisation	Description
value	int	I	m	Nouvelle valeur de la VARIABLE.
<return>	int	O	m	Valeur de la VARIABLE.

6.4.14 Builtin put_date**Profils de communication: FF, ISA100****Objet: Stocker une valeur de date dans la VARIABLE**

Le builtin put_date est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
put_date(data, size)
```

Les éléments lexicaux du builtin put_date sont présentés dans le Tableau 145.

Tableau 145 – Builtin put_date

Nom du paramètre	Type	Direction	Utilisation	Description
data	char[]	I	m	Nouvelle valeur de la VARIABLE.
size	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long date_size, status;
char date_value[7];

/* set DATE_AND_TIME to August 19, 2016 14:30:03 */

date_size = 7;
date_value[0]=0x0B;      /* 3 seconds = 3000 ms = 0xBB8 */
date_value[1]=0xB8;
date_value[2]= 30;       /* 30 minutes */
date_value[3]= 14;       /* 14 hours */
date_value[4]=0xB3;      /* 0xB3 = (5 (Friday) << 5) + 19 (day of month) */
date_value[5]= 8;        /* August = 8 */
date_value[6]= 116;      /* year of 2016 = 116 + 1900 */

status = put_date(date_value, date_size);
```

6.4.15 Builtin put_double

Profils de communication: FF, ISA100

Objet: Stocker une valeur double précision dans la VARIABLE

Le builtin put_double est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

put_double(value)

Les éléments lexicaux du builtin put_double sont présentés dans le Tableau 146.

Tableau 146 – Builtin put_double

Nom du paramètre	Type	Direction	Utilisation	Description
value	double	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
double value;
value = 3.14159;
status = put_double(value);
```

6.4.16 Builtin put_float

Profils de communication: FF, ISA100

Objet: Stocker une valeur à virgule flottante dans la VARIABLE

Le builtin put_float est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

put_float(value)

Les éléments lexicaux du builtin put_float sont présentés dans le Tableau 147.

Tableau 147 – Builtin put_float

Nom du paramètre	Type	Direction	Utilisation	Description
value	double	I	m	Nouvelle valeur de la VARIABLE. La valeur est transmise en tant que valeur double précision pour permettre de transmettre une expression. De nombreux compilateurs réalisent une conversion de simple à double précision avant d'évaluer les expressions
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
value = 279.55025;
status = put_float((double) value);
```

6.4.17 Builtin put_signed**Profils de communication: FF, ISA100****Objet: Stocker une valeur signée dans la VARIABLE**

Le builtin put_signed est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié spécifié dans le Tableau 9.

Structure lexicale

```
put_signed(value)
```

Les éléments lexicaux du builtin put_signed sont présentés dans le Tableau 148.

Tableau 148 – Builtin put_signed

Nom du paramètre	Type	Direction	Utilisation	Description
value	long	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long value;
value = -3592;
status = put_signed(value);
```

6.4.18 Builtin put_string

Profils de communication: FF, ISA100

Objet: Stocker une valeur de chaîne dans la VARIABLE

Le builtin put_string est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Les VARIABLES avec les types de données enumerated, bit_enumerated et index doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C.

Structure lexicale

```
put_string(string, len)
```

Les éléments lexicaux du builtin put_string sont présentés dans le Tableau 149.

Tableau 149 – Builtin put_string

Nom du paramètre	Type	Direction	Utilisation	Description
string	DD_STRING	I	m	Nouvelle valeur de la VARIABLE. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
len	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = put_string("xyz", 3);
```

6.4.19 Builtin put_unsigned

Profils de communication: FF, ISA100

Objet: Stocker une valeur non signée dans la VARIABLE

Le builtin put_unsigned est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
put_unsigned(value)
```

Les éléments lexicaux du builtin put_unsigned sont présentés dans le Tableau 150.

Tableau 150 – Builtin put_unsigned

Nom du paramètre	Type	Direction	Utilisation	Description
value	unsigned long	I	o	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
value = 123;
status = put_unsigned(value);
```

6.4.20 Builtin sgetval

Profils de communication: CS, GPE, HART, PB, PN

Objet: Obtenir la valeur d'une VARIABLE de type chaîne

Le builtin sgetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin retourne la valeur d'une VARIABLE telle qu'elle a été reçue par le dispositif connecté. Des calculs personnalisés peuvent ensuite être effectués sur la valeur de la VARIABLE avant qu'elle ne soit stockée dans l'application EDD. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être récupérée ou si la variable n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

```
sgetval (value)
```

Les éléments lexicaux du builtin sgetval sont présentés dans le Tableau 151.

Tableau 151 – Builtin sgetval

Nom du paramètre	Type	Direction	Utilisation	Description
value	DD_STRING	O	m	Valeur de la VARIABLE. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	int	O	m	BI_SUCCESS si l'assignation a abouti.

Exemple

```
DD_STRING sval;
sgetval (sval);
```

Dans l'exemple ci-dessus (en prenant pour hypothèse qu'il s'agit d'une méthode de prédiction sur la VARIABLE "tag", une variable de 8 caractères), sval est défini sur la valeur actuelle de la VARIABLE "tag".

6.4.21 Builtin ssetval

Profils de communication: CS, GPE, HART, PB, PN

Objet: Définir la valeur d'une VARIABLE de type chaîne

Le builtin ssetval est spécifiquement conçu pour être utilisé dans des actions de VARIABLE. Voir le Tableau 9 pour la liste des types d'actions pouvant appeler ce builtin.

Ce builtin définit la valeur d'une VARIABLE sur la valeur spécifiée. Si la chaîne transmise est plus longue que la taille spécifiée pour la VARIABLE, elle est tronquée de façon à l'adapter. Si la chaîne transmise est plus courte que la taille spécifiée pour la VARIABLE, elle est complétée avec des espaces. La méthode est interrompue s'il ne s'agit pas d'une action appelée depuis une VARIABLE, si la valeur de la VARIABLE ne peut pas être définie ou si la VARIABLE n'est pas du type approprié défini dans le Tableau 9.

Structure lexicale

ssetval (value)

Les éléments lexicaux du builtin ssetval sont présentés dans le Tableau 152.

Tableau 152 – Builtin ssetval

Nom du paramètre	Type	Direction	Utilisation	Description
value	DD_STRING	I	m	Valeur de la VARIABLE. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
<return>	DD_STRING	O	m	Valeur de la VARIABLE.

6.5 Builtins d'accès aux variables

6.5.1 Builtin assign

Profils de communication: FF, ISA100

Objet: Définir la valeur d'un paramètre

Le builtin assign définit la valeur d'une VARIABLE sur la valeur d'une autre VARIABLE. Il lit la valeur de la VARIABLE source (depuis le dispositif ou la table des variables) et définit la valeur de la VARIABLE de destination (dans le dispositif ou la table des variables) sur la valeur de la VARIABLE source.

NOTE Le builtin est un raccourci pour effectuer un get_<type>_value suivi d'un put_<type>_value où <type> est l'un des types de valeurs suivants: date, float, double, signed, string, unsigned.

Les paramètres dst_id, src_id, dst_member_id et src_member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER ID et de la référence de la VARIABLE. La VARIABLE source et la VARIABLE de destination doivent être du même type.

Structure lexicale

assign(dst_id, dst_member_id, src_id, src_member_id)

Les éléments lexicaux du builtin assign sont présentés dans le Tableau 153.

Tableau 153 – Builtin assign

Nom du paramètre	Type	Direction	Utilisation	Description
dst_id	unsigned long	I	m	ID d'élément de la VARIABLE de destination.
dst_member_id	unsigned long	I	m	ID de membre de la VARIABLE de destination.
src_id	unsigned long	I	m	ID d'élément de la VARIABLE source.
src_member_id	unsigned long	I	m	ID de membre de la VARIABLE source.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = assign(ITEM_ID(PARAM.OUT), MEMBER_ID(VALUE),
```

6.5.2 Builtin assign_double**Profils de communication: CS, GPE, HART, PB, PN****Objet: Assigner une valeur à une VARIABLE de type double**

Le builtin assign_double assigne une valeur à une VARIABLE. La VARIABLE référencée doit être une VARIABLE de type DOUBLE.

Structure lexicale

```
assign_double(device_var, new_value)
```

Les éléments lexicaux du builtin assign_double sont présentés dans le Tableau 154.

Tableau 154 – Builtin assign_double

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE.
new_value	double	I	m	Nouvelle valeur de la VARIABLE référencée.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

Exemple

```
assign_double(pressure_upper_range_value, 1.2345678);
```

Dans cet exemple, la VARIABLE pressure_upper_range_value est définie sur la valeur à virgule flottante 1,2345678.

6.5.3 Builtin assign_float

Profils de communication: CS, GPE, HART, PB, PN

Objet: Assigner une valeur à une VARIABLE de type à virgule flottante

Le builtin assign_float assigne une valeur à une VARIABLE. La VARIABLE référencée doit être une VARIABLE de type FLOAT.

Structure lexicale

```
assign_float(device_var, new_value)
```

Les éléments lexicaux du builtin assign_float sont présentés dans le Tableau 155.

Tableau 155 – Builtin assign_float

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE.
new_value	double	I	m	Nouvelle valeur de la VARIABLE référencée.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

Exemple

```
assign_float(pressure_upper_range_value, 1.23);
```

Dans cet exemple, la VARIABLE pressure_upper_range_value est définie sur la valeur à virgule flottante 1,23.

6.5.4 Builtin assign_int

Profils de communication: CS, GPE, HART, PB, PN

Objet: Assigner une valeur à une VARIABLE de type entier

Le builtin assign_int assigne une valeur à une VARIABLE. La VARIABLE référencée doit être une VARIABLE de type INTEGER.

Structure lexicale

```
assign_int(device_var, new_value)
```

Les éléments lexicaux du builtin assign_int sont présentés dans le Tableau 156.

Tableau 156 – Builtin assign_int

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE.
new_value	int	I	m	Nouvelle valeur de la VARIABLE référencée.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

Exemple

```
assign_int(pressure_upper_range_value, 1);
```

Dans cet exemple, la VARIABLE `pressure_upper_range_value` est définie sur la valeur entière 1.

6.5.5 Builtin assign_var

Profils de communication: CS, GPE, HART, PB, PN

Objet: Assigner une valeur d'une VARIABLE à une autre

Le builtin `assign_var` assigne une valeur de la VARIABLE source à la VARIABLE de destination. Les deux VARIABLES doivent être du même type.

Structure lexicale

```
assign_var(destination_var, source_var)
```

Les éléments lexicaux du builtin `assign_var` sont présentés dans le Tableau 157.

Tableau 157 – Builtin assign_var

Nom du paramètre	Type	Direction	Utilisation	Description
<code>destination_var</code>	reference	I	m	Nom de la VARIABLE de destination.
<code>source_var</code>	reference	I	m	Nom de la VARIABLE source.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

Exemple

```
vassign(pressure_units, range_units);
```

Dans cet exemple, la valeur actuelle de `range_units` est stockée dans la valeur actuelle de `pressure_units`.

6.5.6 Builtin assign2

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `assign2` lit une valeur dans le bloc source et l'écrit dans le bloc de destination. Dans une application EDD en mémoire cache, la valeur source est extraite de la table de variables du bloc source et stockée dans la table de variables du bloc de destination, tandis que dans une application EDD non en mémoire cache, la valeur source est lue depuis le bloc source dans le dispositif et écrite dans le bloc de destination dans le dispositif.

Les paramètres `dst_blk_id`, `dst_blk_num`, `dst_id` et `dst_member_id` spécifient collectivement la VARIABLE, le membre d'enregistrement ou l'élément de matrice de destination, tandis que les paramètres `src_blk_id`, `src_blk_num`, `src_id` et `src_member_id` spécifient collectivement la VARIABLE, le membre d'enregistrement ou l'élément de matrice source.

Les paramètres `dst_blk_id` et `src_blk_id` spécifient l'ID d'élément du bloc auquel accéder, tandis que les paramètres `dst_blk_num` et `src_blk_num` spécifient les indices relatifs du bloc auquel accéder.

Si le paramètre `dst_id` référence une VARIABLE simple, le paramètre `dst_member_id` est ignoré. Si le paramètre `dst_id` référence un enregistrement, le paramètre `dst_member_id` spécifie un membre de l'enregistrement. Si le paramètre `dst_id` référence un paramètre de matrice, le paramètre `dst_member_id` spécifie un élément de la matrice.

Si le paramètre `src_id` référence une VARIABLE simple, le paramètre `src_member_id` est ignoré. Si le paramètre `src_id` référence un enregistrement, le paramètre `src_member_id` spécifie un membre de l'enregistrement. Si le paramètre `src_id` référence un paramètre de matrice, le paramètre `src_member_id` spécifie un élément de la matrice.

Les paramètres `dst_blk_id`, `src_blk_id`, `dst_id`, `src_id`, `dst_member_id` et `src_member_id` sont généralement spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Les paramètres `dst_blk_num` et `src_blk_num` sont généralement spécifiés à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

NOTE La fonction intégrée fournit un raccourci pour effectuer un `get_<type>_value2()` suivi d'un `put_<type>_value2()`, où `<type>` est l'un des types suivants: `date`, `float`, `double`, `signed`, `string`, `unsigned`.

La valeur source et la valeur de destination doivent être du même type.

Structure lexicale

```
assign2(dst_blk_id, dst_blk_num, dst_id, dst_member_id, src_blk_id,
src_blk_num, src_id, src_member_id)
```

Les éléments lexicaux du builtin `assign2` sont présentés dans le Tableau 158.

Tableau 158 – Builtin assign2

Nom du paramètre	Type	Direction	Utilisation	Description
<code>dst_blk_id</code>	unsigned long	I	m	ID d'élément du bloc de destination.
<code>dst_blk_num</code>	unsigned long	I	m	Numéro d'occurrence du bloc de destination.
<code>dst_id</code>	unsigned long	I	m	ID d'élément de la VARIABLE, de l'enregistrement ou de la matrice de destination.
<code>dst_member_id</code>	unsigned long	I	m	ID de membre du membre d'enregistrement ou de l'élément de matrice de destination.
<code>src_blk_id</code>	unsigned long	I	m	ID d'élément du bloc source.
<code>src_blk_num</code>	unsigned long	I	m	Numéro d'occurrence du bloc source.
<code>src_id</code>	unsigned long	I	m	ID d'élément de la VARIABLE, de l'enregistrement ou de la matrice source.
<code>src_member_id</code>	unsigned long	I	m	ID de membre du membre d'enregistrement ou de l'élément de matrice source.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long aiBlock;
aiBlock = ITEM_ID(__analog_input_block);
status = assign2(aiBlock, 0, MEMBER_ID(OUT), MEMBER_ID(VALUE),
aiBlock, 0, MEMBER_ID(PV), MEMBER_ID(VALUE));
```

6.5.7 Builtin dassign**Profils de communication: CS, GPE, HART, PB, PN****Objet: Assigner une valeur à une VARIABLE de type double**

Le builtin dassign assigne une valeur à une VARIABLE. La VARIABLE référencée doit être une VARIABLE de type DOUBLE.

Structure lexicale

```
dassign(device_var, new_value)
```

Les éléments lexicaux du builtin dassign sont présentés dans le Tableau 159.

Tableau 159 – Builtin dassign

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE.
new_value	double	I	m	Nouvelle valeur de la VARIABLE référencée.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

6.5.8 Builtin fassign**Profils de communication: CS, GPE, HART, PB, PN****Objet: Assigner une valeur à une VARIABLE de type à virgule flottante**

Le builtin fassign assigne une valeur à une VARIABLE. La VARIABLE référencée doit être une VARIABLE de type FLOAT.

Structure lexicale

```
fassign(device_var, new_value)
```

Les éléments lexicaux du builtin fassign sont présentés dans le Tableau 160.

Tableau 160 – Builtin fassign

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE.
new_value	double	I	m	Nouvelle valeur de la VARIABLE référencée.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

6.5.9 Builtin float_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Obtenir la valeur d'une VARIABLE de type à virgule flottante

Le builtin float_value retourne la valeur de la VARIABLE spécifiée. La VARIABLE référencée doit être de type FLOAT.

Structure lexicale

```
float_value(source_var_name)
```

Les éléments lexicaux du builtin float_value sont présentés dans le Tableau 161.

Tableau 161 – Builtin float_value

Nom du paramètre	Type	Direction	Utilisation	Description
source_var_name	reference	I	m	Nom d'une référence de VARIABLE.
<return>	double	O	m	Valeur de la VARIABLE.

Exemple

```
float fval;
fval = float_value(pressure_upper_range_value);
```

Dans cet exemple, la valeur de pressure_upper_range_value est retournée et stockée dans le paramètre fval de la variable de méthode.

6.5.10 Builtin fvar_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Obtenir la valeur d'une VARIABLE de type à virgule flottante

Le builtin fvar_value retourne la valeur de la VARIABLE spécifiée. La VARIABLE référencée doit être de type FLOAT.

Structure lexicale

```
fvar_value(source_var)
```

Les éléments lexicaux du builtin fvar_value sont présentés dans le Tableau 162.

Tableau 162 – Builtin fvar_value

Nom du paramètre	Type	Direction	Utilisation	Description
source_var	reference	I	m	Nom de la référence de VARIABLE.
<return>	double	O	m	Valeur de la VARIABLE.

6.5.11 Builtin get_date_value

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE de date

Le builtin get_date_value extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

La valeur est une mémoire tampon d'octet qui peut contenir jusqu'à huit caractères et une VARIABLE de type date-and-time, time ou duration du dispositif.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE en question.

Structure lexicale

```
get_date_value(id, member_id, data, size)
```

Les éléments lexicaux du builtin get_date_value sont présentés dans le Tableau 163.

Tableau 163 – Builtin get_date_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'une référence de VARIABLE.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
data	char[]	O	m	Mémoire tampon de 8 octets max. pour le résultat du builtin.
size	long	O	m	Taille de la mémoire tampon utilisée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status, date_size;
char date_buf[8];
```

```
status = get_date_value(ITEM_ID(PARAM.CALIBRATION), MEMBER_ID(DATE),
date_buf, &date_size);
```

6.5.12 Builtin get_date_value2

Profils de communication: FF, ISA100

Objet: Extraire la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin get_date_value2 extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

La valeur est une mémoire tampon d'octets qui peut contenir jusqu'à huit caractères et une VARIABLE de type date-and-time, time ou duration du dispositif.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre id référence une VARIABLE simple, le paramètre member_id est ignoré. Si le paramètre id référence un enregistrement, le paramètre member_id spécifie un membre de l'enregistrement. Si le paramètre id référence un paramètre de matrice, le paramètre member_id spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont généralement spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID.

Le paramètre blk_num est généralement spécifié à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

Les paramètres value et length spécifient une mémoire tampon qui, après retour, contient jusqu'à huit caractères du type de données Date, Time of Day ou Time Difference.

Le type de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice doit être l'un des suivants:

- DATE_AND_TIME
- TIME
- DURATION

Structure lexicale

get_date_value2(blk_id, blk_num, id, member_id, data, size)

Les éléments lexicaux du builtin get_date_value2 sont présentés dans le Tableau 164.

Tableau 164 – Builtin get_date_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	m	ID d'une référence de VARIABLE, RECORD ou ARRAY.
member_id	unsigned long	I	m	Membre d'enregistrement ou élément de matrice auquel accéder.
data	char[]	O	m	Mémoire tampon de 8 octets max. pour le résultat du builtin.
size	long	O	m	Taille de la mémoire tampon utilisée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status, date_size;
char date_buf[8];

status = get_date_value2(ITEM_ID(transducer_block), 0,
MEMBER_ID(CALIBRATION), MEMBER_ID(DATE), date_buf, &date_size);
```

6.5.13 Builtin get_double_value

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE double à virgule flottante

Le builtin get_double_value extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE.

La variable doit avoir un type de données DOUBLE.

Structure lexicale

```
get_double_value(id, member_id, value)
```

Les éléments lexicaux du builtin `get_double_value` sont présentés dans le Tableau 165.

Tableau 165 – Builtin `get_double_value`

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	double	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
double value;
status = get_double_value(ITEM_ID(PARAM.SAMPLE), 0, &value);
```

6.5.14 Builtin `get_double_value2`

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `get_double_value2` extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice spécifié et la retourne.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la VARIABLE. Le paramètre `blk_num` est généralement spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

La VARIABLE, le membre d'enregistrement ou la matrice doit avoir un type de données DOUBLE.

Structure lexicale

```
get_double_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin `get_double_value2` sont présentés dans le Tableau 166.

Tableau 166 – Builtin get_double_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	double	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
double value;
status = get_double_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, &value);
```

6.5.15 Builtin get_float_value

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle mise à l'échelle d'une VARIABLE à virgule flottante

Le builtin get_float_value extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE.

La variable doit avoir un type de données FLOAT.

Structure lexicale

```
get_float_value(id, member_id, value)
```

Les éléments lexicaux du builtin get_float_value sont présentés dans le Tableau 167.

Tableau 167 – Builtin get_float_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	float	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
status = get_float_value(ITEM_ID(PARAM.SAMPLE), 0, &value);
```

6.5.16 Builtin get_float_value2**Profils de communication: FF, ISA100**

Objet: Extraire la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `get_float_value2` extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la VARIABLE. Le paramètre `blk_num` est généralement spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Le type de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice doit être `FLOAT`.

Structure lexicale

```
get_float_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin `get_float_value2` sont présentés dans le Tableau 168.

Tableau 168 – Builtin get_float_value2

Nom du paramètre	Type	Direction	Utilisation	Description
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance <code>BLOCK_A</code> à laquelle accéder.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de <code>BLOCK_A</code> à laquelle accéder.
<code>id</code>	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
<code>value</code>	float	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
status = get_float_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, &value);
```

6.5.17 Builtin get_signed_value

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle mise à l'échelle d'une VARIABLE entière signée

Le builtin get_signed_value extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

L'ID et le member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE. La VARIABLE doit avoir un type de données signed long.

Structure lexicale

```
get_signed_value(id, member_id, value)
```

Les éléments lexicaux du builtin get_signed_value sont présentés dans le Tableau 169.

Tableau 169 – Builtin get_signed_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	signed long	O	m	Valeur retournée de la VARIABLE signée.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long value;
status = get_signed_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(DECIMAL_PT),
&value);
```

6.5.18 Builtin get_signed_value2

Profils de communication: FF, ISA100

Objet: Extraire la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin get_signed_value2 extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice spécifié et la retourne.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice doit avoir un type de données signed long.

Structure lexicale

```
get_signed_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin `get_signed_value2` sont présentés dans le Tableau 170.

Tableau 170 – Builtin `get_signed_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
<code>id</code>	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
<code>value</code>	signed long	O	m	Valeur retournée de la VARIABLE signée.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long value;
status = get_signed_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_SCALE), MEMBER_ID(DECIMAL_PT), &value);
```

6.5.19 Builtin `get_string_value`

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE de chaîne

Le builtin `get_string_value` extrait de la mémoire cache ou du dispositif la valeur de chaîne actuelle de la VARIABLE spécifiée et la retourne.

Les paramètres `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la variable en question.

La VARIABLE doit avoir un type de données char et être une mémoire tampon d'octet qui peut contenir différents types de données non-ANSI, tels que:

- VISIBLE
- OCTET

- PASSWORD
- BITSTRING
- EUC
- ASCII

Structure lexicale

```
get_string_value(id, member_id, string, len)
```

Les éléments lexicaux du builtin `get_string_value` sont présentés dans le Tableau 171.

Tableau 171 – Builtin `get_string_value`

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'une référence de VARIABLE.
member_id	unsigned long	I	m	ID de membre d'un BLOCK_A, PARAMETER_LIST, RECORD, VALUE_ARRAY, COLLECTION ou REFERENCE_ARRAY.
string	char*	O	m	Mémoire tampon de la chaîne de résultat.
len	long	O	m	Taille de la chaîne de résultat.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long len;
char buf[256];
len = 256;
status = get_string_value(ITEM_ID(PARAM.TAG_DESC), 0, buf, &len);
```

6.5.20 Builtin `get_string_value2`

Profils de communication: FF, ISA100

Objet: Extraire la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `get_string_value2` extrait de la mémoire cache ou du dispositif la valeur de chaîne actuelle de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice spécifié et la retourne.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

Le type de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice doit être l'un des suivants:

- EUC
- ASCII
- PASSWORD
- BITSTRING
- VISIBLE
- OCTET

Structure lexicale

```
get_string_value2(blk_id, blk_num, id, member_id, string, len)
```

Les éléments lexicaux du builtin `get_string_value2` sont présentés dans le Tableau 172.

Tableau 172 – Builtin `get_string_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	m	ID d'une référence de VARIABLE.
member_id	unsigned long	I	m	ID de membre d'un BLOCK_A, PARAMETER_LIST, RECORD, VALUE_ARRAY, COLLECTION ou REFERENCE_ARRAY.
string	char*	O	m	Spécifie la mémoire tampon de la chaîne de résultat.
len	long	O	m	Taille de la chaîne de résultat.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
long len;
char buf[256];
len = 256;
status = get_string_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(TAG_DESC), 0, buf, &len);
```

6.5.21 Builtin `get_unsigned_value`

Profils de communication: FF, ISA100

Objet: Obtenir la valeur actuelle d'une VARIABLE entière non signée

Le builtin `get_unsigned_value` extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE spécifiée et la retourne.

Les paramètres `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la VARIABLE en question.

La VARIABLE doit avoir un type de données unsigned. Une valeur non signée peut contenir des VARIABLES des types de données suivants:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

Les VARIABLES avec les types de données enumerated, bit_enumerated et index doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C.

Structure lexicale

`get_unsigned_value(id, member_id, value)`

Les éléments lexicaux du builtin `get_unsigned_value` sont présentés dans le Tableau 173.

Tableau 173 – Builtin `get_unsigned_value`

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	unsigned long	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
status = get_unsigned_value(ITEM_ID(PARAM.PV_STATE), 0, &value);
```

6.5.22 Builtin `get_unsigned_value2`

Profils de communication: FF, ISA100

Objet: Extraire la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `get_unsigned_value2` extrait de la mémoire cache ou du dispositif la valeur actuelle mise à l'échelle de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice spécifié et la retourne.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blockInstance` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Le paramètre blk_num est spécifié à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

La VARIABLE doit avoir un type de données unsigned. Une valeur non signée peut contenir des VARIABLES des types de données suivants:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

Les VARIABLES avec les types de données enumerated, bit_enumerated et index doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C.

Structure lexicale

```
get_unsigned_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin get_unsigned_value2 sont présentés dans le Tableau 174.

Tableau 174 – Builtin get_unsigned_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	m	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	m	ID de membre de la VARIABLE à laquelle accéder.
value	unsigned long	O	m	Valeur retournée de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
status = get_unsigned_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_STATE), 0, &value);
```

6.5.23 Builtin iassign

Profils de communication: CS, GPE, HART, PB, PN

Objet: Assigner une valeur à une VARIABLE de type entier

Le builtin iassign assigne la valeur spécifiée à la VARIABLE. La VARIABLE doit référencer une VARIABLE de type INTEGER.

Structure lexicale

```
iassign(device_var, new_value)
```

Les éléments lexicaux du builtin iassign sont présentés dans le Tableau 175.

Tableau 175 – Builtin iassign

Nom du paramètre	Type	Direction	Utilisation	Description
device_var	reference	I	m	Nom de la VARIABLE à laquelle accéder.
new_value	int	I	m	Nouvelle valeur de la VARIABLE.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377. BI_SUCCESS si l'assignation a abouti et BI_ERROR si l'identifiant de la VARIABLE était non valide ou incorrect.

6.5.24 Builtin int_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Obtenir la valeur d'une VARIABLE de type entier

Le builtin int_value retourne la valeur de la VARIABLE spécifiée. La VARIABLE doit être de type INTEGER.

Structure lexicale

int_value(variable)

Les éléments lexicaux du builtin int_value sont présentés dans le Tableau 176.

Tableau 176 – Builtin int_value

Nom du paramètre	Type	Direction	Utilisation	Description
variable	reference	I	m	VARIABLE dont la valeur doit être extraite.
<return>	int	O	m	Valeur de la VARIABLE.

6.5.25 Builtin ivar_value

Profils de communication: CS, GPE, HART, PB, PN

Objet: Obtenir la valeur d'une VARIABLE de type entier

Le builtin ivar_value retourne la valeur de la VARIABLE spécifiée. Le paramètre source_var_name de la VARIABLE doit être de type INTEGER.

Structure lexicale

ivar_value(source_var_name)

Les éléments lexicaux du builtin ivar_value sont présentés dans le Tableau 177.

Tableau 177 – Builtin ivar_value

Nom du paramètre	Type	Direction	Utilisation	Description
source_var_name	reference	I	m	Nom de la VARIABLE.
<return>	int	O	m	Valeur de la VARIABLE.

6.5.26 Builtin lassign**Profils de communication: CS, GPE, PB, PN****Objet: Assigner une valeur longue à une VARIABLE**

Le builtin lassign assigne la valeur spécifiée à la VARIABLE. La VARIABLE doit référencer une VARIABLE de type long.

Structure lexicale

```
lassign(var_name, new_value)
```

Les éléments lexicaux du builtin lassign sont présentés dans le Tableau 178.

Tableau 178 – Builtin lassign

Nom du paramètre	Type	Direction	Utilisation	Description
var_name	reference	I	m	Nom de la VARIABLE.
new_value	long	I	m	Nouvelle valeur de la VARIABLE.
<return>	int	O	m	Un des codes de retour spécifiés dans le Tableau 377.

6.5.27 Builtin long_value**Profils de communication: CS, GPE, PB, PN****Objet: Obtenir la valeur longue d'une VARIABLE**

Le builtin long_value retourne la valeur de la VARIABLE spécifiée. Le paramètre source_var_name de la VARIABLE doit référencer une VARIABLE de type long.

Structure lexicale

```
long_value(source_var_name)
```

Les éléments lexicaux du builtin long_value sont présentés dans le Tableau 179.

Tableau 179 – Builtin long_value

Nom du paramètre	Type	Direction	Utilisation	Description
source_var_name	reference	I	m	Nom de la VARIABLE.
<return>	int	O	m	Valeur de la VARIABLE.

6.5.28 Builtin lvar_value**Profils de communication: CS, GPE, PB, PN****Objet: Obtenir une valeur longue**

Le builtin lvar_value retourne la valeur de la VARIABLE spécifiée.

Structure lexicale

```
lvar_value(source_var_name)
```

Les éléments lexicaux du builtin lvar_value sont présentés dans le Tableau 180.

Tableau 180 – Builtin lvar_value

Nom du paramètre	Type	Direction	Utilisation	Description
source_var_name	reference	I	m	Nom de la VARIABLE.
<return>	long	O	m	Valeur de la VARIABLE.

6.5.29 Builtin put_date_value

Profils de communication: FF, ISA100

Objet: Placer la valeur spécifiée d'une variable date_and_time, time ou duration

Le builtin put_date_value stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE en question.

Le builtin place la valeur spécifiée de la VARIABLE date_and_time, time ou duration dans la mémoire cache ou dans le dispositif.

La VARIABLE doit avoir un type de données char. La chaîne de caractères peut contenir jusqu'à huit caractères et contenir une VARIABLE des types de données suivants:

- DATE_AND_TIME
- TIME
- DURATION

Structure lexicale

put_date_value(id, member_id, data, size)

Les éléments lexicaux du builtin put_date_value sont présentés dans le Tableau 181.

Tableau 181 – Builtin put_date_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
data	char[]	I	m	Nouvelle valeur de la VARIABLE.
size	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long duration_size, status;
char duration_value[6];

/* set DURATION to 25 (days): 10 (hours): 20: 05.218 */

duration_size = 6;
duration_value[0] = 0x02;    /*0x0237B4E2 = 37205218 ms since midnight */
duration_value[1] = 0x37;
duration_value[2] = 0xB4;
duration_value[3] = 0xE2;
duration_value[4] = 0x00;    /*0x0019 = 25 days */
duration_value[5] = 0x19;

status = put_date_value (ITEM_ID (PARAM.CALIBRATION), MEMBER_ID (DURATION),
duration_value, duration_size);

```

6.5.30 Builtin put_date_value2**Profils de communication: FF, ISA100****Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice**

Le builtin put_date_value2 stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre id référence une VARIABLE simple, le paramètre member_id est ignoré. Si le paramètre id référence un enregistrement, le paramètre member_id spécifie un membre de l'enregistrement. Si le paramètre id référence un paramètre de matrice, le paramètre member_id spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Le paramètre blk_num est spécifié à l'aide des fonctions intégrées get_block_instance_by_object_index et get_block_instance_by_block_tag des builtins.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice doit avoir un type de données char. La chaîne de caractères peut contenir jusqu'à huit caractères et contenir une VARIABLE des types de données suivants:

- DATA_AND_TIME
- TIME
- DURATION

Structure lexicale

```
put_date_value2(blk_id, blk_num, id, member_id, data, size)
```

Les éléments lexicaux du builtin put_date_value2 sont présentés dans le Tableau 182.

Tableau 182 – Builtin put_date_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
data	char[]	I	m	Nouvelle valeur de la VARIABLE.
size	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status, time_size;
char time[6];
```

```
/* set TIME to Oct 14, 2010 03:03:03 */
```

```
time_size = 6;
time[0]=0;          /* 0xA79658 = 10983000 ms since midnight */
time[1]=0xA7;
time[2]=0x96;
time[3]=0x58;
time[4]=0x26; /* 0x2637 = 9783 days since January 1, 1984 */
time[5]=0x37;
```

```
status = put_date_value2(ITEM_ID(transducer_block), 0,
MEMBER_ID(CALIBRATION), MEMBER_ID(TIME), time, time_size);
```

6.5.31 Builtin put_double_value**Profils de communication: FF, ISA100****Objet: Placer la valeur spécifiée mise à l'échelle d'une VARIABLE double**

Le builtin put_double_value stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE en question.

La variable doit avoir un type de données DOUBLE.

Structure lexicale

```
put_double_value(id, member_id, value)
```

Les éléments lexicaux du builtin put_double_value sont présentés dans le Tableau 183.

Tableau 183 – Builtin put_double_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
value	double	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
double value;
value = 100090.738;
status = put_double_value(ITEM_ID(PARAM.SAMPLE), 0, value);
```

6.5.32 Builtin put_double_value2**Profils de communication: FF, ISA100****Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice**

Le builtin put_double_value2 stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle, du membre d'enregistrement ou de l'élément de matrice dans la mémoire cache ou dans le dispositif.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre id référence une VARIABLE simple, le paramètre member_id est ignoré. Si le paramètre id référence un enregistrement, le paramètre member_id spécifie un membre de l'enregistrement. Si le paramètre id référence un paramètre de matrice, le paramètre member_id spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Le paramètre blk_num est spécifié à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice doit avoir un type de données DOUBLE.

Structure lexicale

```
put_double_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin put_double_value2 sont présentés dans le Tableau 184.

Tableau 184 – Builtin put_double_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
value	double	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```

long status;
double value;
value = 1000090.738;
status = put_double_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, value);

```

6.5.33 Builtin put_float_value**Profils de communication: FF, ISA100****Objet: Placer la valeur spécifiée d'une VARIABLE à virgule flottante**

Le builtin put_float_value stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Le type de valeur est double précision afin de permettre la promotion automatique qui peut se produire lorsque deux valeurs à virgule flottante sont multipliées.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE en question.

La variable doit avoir un type de données FLOAT.

Structure lexicale

```
put_float_value(id, member_id, value)
```

Les éléments lexicaux du builtin put_float_value sont présentés dans le Tableau 185.

Tableau 185 – Builtin put_float_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
value	double	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
value = 57.2958;
status = put_float_value2(ITEM_ID(PARAM.SAMPLE), 0, (double) value);
```

6.5.34 Builtin put_float_value2**Profils de communication: FF, ISA100**

Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin put_float_value2 stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle, du membre d'enregistrement ou de l'élément de matrice dans la mémoire cache ou dans le dispositif.

Le type de valeur est double précision afin de permettre la promotion automatique qui peut se produire lorsque deux valeurs à virgule flottante sont multipliées.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre id référence une VARIABLE simple, le paramètre member_id est ignoré. Si le paramètre id référence un enregistrement, le paramètre member_id spécifie un membre de l'enregistrement. Si le paramètre id référence un paramètre de matrice, le paramètre member_id spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Le paramètre blk_num est spécifié à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

La variable doit avoir un type de données FLOAT.

Structure lexicale

```
put_float_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin put_float_value2 sont présentés dans le Tableau 186.

Tableau 186 – Builtin put_float_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
value	double	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
float value;
value = 57.2958;
status = put_float_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(SAMPLE), 0, (double) value);
```

6.5.35 Builtin put_signed_value

Profils de communication: FF, ISA100

Objet: Placer la valeur spécifiée d'une VARIABLE entière signée

Le builtin put_signed_value stocke la nouvelle valeur de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Les paramètres id et member_id sont spécifiés en utilisant les builtins ITEM_ID et MEMBER_ID et le nom ou la référence de la VARIABLE en question.

La VARIABLE à écrire doit avoir un type de données long.

Structure lexicale

```
put_signed_value(id, member_id, value)
```

Les éléments lexicaux du builtin put_signed_value sont présentés dans le Tableau 187.

Tableau 187 – Builtin put_signed_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
value	long	I	m	Nouvelle valeur de la VARIABLE.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = put_signed_value(ITEM_ID(PARAM.PV_SCALE), MEMBER_ID(DECIMAL_PT),
3);
```

6.5.36 Builtin put_signed_value2

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin put_signed_value2 stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle, du membre d'enregistrement ou de l'élément de matrice dans la mémoire cache ou dans le dispositif.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice à écrire doit avoir un type de données long.

Structure lexicale

```
put_signed_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin `put_signed_value2` sont présentés dans le Tableau 188.

Tableau 188 – Builtin `put_signed_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
<code>id</code>	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
<code>value</code>	long	I	m	Nouvelle valeur de la VARIABLE.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = put_signed_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_SCALE), MEMBER_ID(DECIMAL_PT), 3);
```

6.5.37 Builtin `put_string_value`

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE de chaîne

Le builtin `put_string_value` stocke la valeur fournie de la VARIABLE spécifiée dans la mémoire cache ou dans le dispositif.

Les paramètres `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la VARIABLE en question.

La VARIABLE doit avoir un type de données char; une chaîne de caractères peut contenir des VARIABLES des types de données suivants:

- ASCII
- PASSWORD
- EUC (unicode étendu)

- BITSTRING
- VISIBLE
- OCTET

Les VARIABLES avec les types de données enumerated, bit_enumerated et index doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C.

Structure lexicale

```
put_string_value(id, member_id, string, len)
```

Les éléments lexicaux du builtin put_string_value sont présentés dans le Tableau 189.

Tableau 189 – Builtin put_string_value

Nom du paramètre	Type	Direction	Utilisation	Description
id	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
string	DD_STRING	I	m	Nouvelle valeur de la VARIABLE. Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
len	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = put_string_value(ITEM_ID(PARAM.TAG_DESC), 0, "|en|PID-101", 11);
```

6.5.38 Builtin put_string_value2

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin put_string_value2 stocke la valeur fournie de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice spécifié dans la mémoire cache ou dans le dispositif.

Le paramètre blk_id spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre blk_num spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre id référence une VARIABLE simple, le paramètre member_id est ignoré. Si le paramètre id référence un enregistrement, le paramètre member_id spécifie un membre de l'enregistrement. Si le paramètre id référence un paramètre de matrice, le paramètre member_id spécifie un élément de la matrice.

Les paramètres blk_id, id et member_id sont spécifiés à l'aide des builtins ITEM_ID et MEMBER_ID. Le paramètre blk_num est spécifié à l'aide des builtins get_block_instance_by_object_index et get_block_instance_by_block_tag.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice doit avoir un type de données char; une chaîne de caractères peut contenir des VARIABLES des types de données suivants:

- ASCII
- PASSWORD
- EUC (unicode étendu)
- BITSTRING
- VISIBLE
- OCTET

Les VARIABLES, membres d'enregistrement ou éléments de matrice avec les types de données enumerated, bit_enumerated et index doivent être gérés par la méthode, car ils ne sont pas pris en charge par le langage de programmation C.

Structure lexicale

```
put_string_value2(blk_id, blk_num, id, member_id, string, len)
```

Les éléments lexicaux du builtin put_string_value2 sont présentés dans le Tableau 190.

Tableau 190 – Builtin put_string_value2

Nom du paramètre	Type	Direction	Utilisation	Description
blk_id	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
blk_num	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
id	unsigned long	I	o	ID d'élément de la VARIABLE, de l'enregistrement ou de la matrice auquel accéder.
member_id	unsigned long	I	o	ID de membre de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice auquel accéder.
string	DD_STRING	I	m	Nouvelle valeur de la VARIABLE, du membre d'enregistrement ou de l'élément de matrice Des types de chaînes autres que DD_STRING peuvent également être utilisés et sont automatiquement convertis en DD_STRING (voir 6.18.3).
len	long	I	m	Longueur de la nouvelle valeur.
<return>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
status = put_string_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(TAG_DESC), 0, "|en|PID-101", 11);
```

6.5.39 Builtin put_unsigned_value

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE non signée

Le builtin put_unsigned_value stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Les paramètres `id` et `member_id` sont spécifiés en utilisant les builtins `ITEM_ID` et `MEMBER_ID` et le nom ou la référence de la VARIABLE en question.

La VARIABLE doit avoir un type de données unsigned. Une valeur non signée peut contenir des VARIABLES des types de données suivants:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

Les VARIABLES de ces types de données doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C. La variable doit être valide dans la mémoire cache ou dans le dispositif.

Structure lexicale

```
put_unsigned_value(id, member_id, value)
```

Les éléments lexicaux du builtin `put_unsigned_value` sont présentés dans le Tableau 191.

Tableau 191 – Builtin `put_unsigned_value`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>id</code>	unsigned long	l	o	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	l	o	ID de membre de la VARIABLE à laquelle accéder.
<code>value</code>	unsigned long	l	o	Nouvelle valeur de la VARIABLE.
<code><return></code>	long	o	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
value = 0xFEEE;
status = put_unsigned_value(ITEM_ID(PARAM.PV_STATE), 0, value);
```

6.5.40 Builtin `put_unsigned_value2`

Profils de communication: FF, ISA100

Objet: Définir la valeur d'une VARIABLE, d'un membre d'enregistrement ou d'un élément de matrice

Le builtin `put_unsigned_value2` stocke la valeur fournie de la VARIABLE spécifiée mise à l'échelle dans la mémoire cache ou dans le dispositif.

Le paramètre `blk_id` spécifie l'ID d'élément du bloc auquel accéder, tandis que le paramètre `blk_num` spécifie le numéro d'occurrence du bloc auquel accéder.

Si le paramètre `id` référence une VARIABLE simple, le paramètre `member_id` est ignoré. Si le paramètre `id` référence un enregistrement, le paramètre `member_id` spécifie un membre de l'enregistrement. Si le paramètre `id` référence un paramètre de matrice, le paramètre `member_id` spécifie un élément de la matrice.

Les paramètres `blk_id`, `id` et `member_id` sont spécifiés à l'aide des builtins `ITEM_ID` et `MEMBER_ID`. Le paramètre `blk_num` est spécifié à l'aide des builtins `get_block_instance_by_object_index` et `get_block_instance_by_block_tag`.

La VARIABLE, le membre d'enregistrement ou l'élément de matrice doit avoir un type de données unsigned. Une valeur non signée peut contenir des VARIABLES des types de données suivants:

- UNSIGNED
- ENUMERATED
- BIT_ENUMERATED
- INDEX
- BOOLEAN

Les VARIABLES de ces types de données doivent être gérées par la méthode, car elles ne sont pas prises en charge par le langage de programmation C.

Structure lexicale

```
put_unsigned_value2(blk_id, blk_num, id, member_id, value)
```

Les éléments lexicaux du builtin `put_unsigned_value2` sont présentés dans le Tableau 192.

Tableau 192 – Builtin `put_unsigned_value2`

Nom du paramètre	Type	Direction	Utilisation	Description
<code>blk_id</code>	unsigned long	I	m	ID d'élément de l'instance BLOCK_A à laquelle accéder.
<code>blk_num</code>	unsigned long	I	m	Numéro d'occurrence de l'instance de BLOCK_A à laquelle accéder.
<code>id</code>	unsigned long	I	o	ID d'élément de la VARIABLE à laquelle accéder.
<code>member_id</code>	unsigned long	I	o	ID de membre de la VARIABLE à laquelle accéder.
<code>value</code>	unsigned long	I	o	Nouvelle valeur de la VARIABLE.
<code><return></code>	long	O	m	Un des codes de retour spécifiés dans le Tableau 376.

Exemple

```
long status;
unsigned long value;
value = 0xFEEE;
status = put_unsigned_value2(ITEM_ID(__analog_input_block), 0,
MEMBER_ID(PV_STATE), 0, value);
```