

Comau Robotics Instruction Handbook



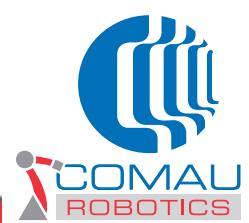
comau.com/robotics

PDL2

Programming Language Manual System Software Rel. 1.17.011

Language Components, Program Structure, Data Representation, Motion Control, Execution Control, Ports, Statements List, Routines, Condition Handlers, Communication, Built-in Routines, Predefined Variables

CR00757609_en-11/2012.12



Le informazioni contenute in questo manuale sono di proprietà di COMAU S.p.A.

E' vietata la riproduzione, anche parziale, senza preventiva autorizzazione scritta di COMAU S.p.A.

COMAU si riserva il diritto di modificare, senza preavviso, le caratteristiche del prodotto presentato in questo manuale.

Copyright © 2008 by COMAU - Pubblicato in data 11/2012

SUMMARY

PREFACE28
Symbols used in the manual	28
Reference documents	29
Modification History	30
1. GENERAL SAFETY PRECAUTIONS32
Responsibilities	32
Safety Precautions	33
Purpose	33
Definitions	33
Applicability	34
Operating Modes	35
Performance	41
2. INTRODUCTION TO PDL242
Peculiarities of PDL2 Language	42
Syntax notation	43
Language components	44
Character set	44
Reserved words, Symbols, and Operators	44
Predefined Identifiers	46
User-defined Identifiers	47
Statements	48
Blank space	48
Comments	48
Program structure	49
Program example	49
Units of measure	50
3. DATA REPRESENTATION51
Data Types	51
INTEGER	52
REAL	52
BOOLEAN	53
STRING	53
ARRAY	54
RECORD	55
VECTOR	56
POSITION	56

Frames of reference	59
JOINTPOS	60
XTNDPOS	60
NODE	61
PATH	61
SEMAPHORE	62
Declarations	62
CONSTANT declarations	63
TYPE declarations	63
VARIABLE declarations	65
Shared types, variables and routines	67
EXPORTED FROM clause	67
GLOBAL attribute and IMPORT statement	68
Expressions	69
Arithmetic Operations	70
Relational Operations	71
Logical Operations	71
Bitwise Operations	72
VECTOR Operations	73
POSITION Operations	73
Data Type conversion	75
Operator precedence	75
Assignment Statement	76
Typecasting	77
4. MOTION CONTROL	79
MOVE Statement	79
ARM Clause	80
TRAJECTORY Clause	80
DESTINATION Clause	81
MOVE TO	81
VIA Clause	82
MOVE NEAR	82
MOVE AWAY	82
MOVE RELATIVE	83
MOVE ABOUT	84
MOVE BY	84
MOVE FOR	84
Optional Clauses	85
ADVANCE Clause	85
TIL Clause	86
WITH Clause	86
SYNCMOVE Clause	87
Continuous motion (MOVEFLY)	88
Timing and Synchronization considerations	89
FLY_CART motion control	90
Motion along a PATH	90
ARM Clause	95
NODE RANGE Clause	95

Optional Clauses	95
ADVANCE Clause	96
WITH Clause	96
Continuous Motion (MOVEFLY)	97
Stopping and Restarting motions	97
CANCEL MOTION Statements	98
LOCK, UNLOCK, and RESUME Statements	98
SIGNAL SEGMENT Statement	99
HOLD Statement	99
ATTACH and DETACH Statements	99
HAND Statements	100
5. PORTS	102
General	102
User-defined and Appl-defined Ports	104
\$DIN and \$DOUT	104
\$IN and \$OUT	104
\$AIN and \$AOUT	104
\$FMI and \$FMO	104
System-defined Ports	105
\$SDI and \$SDO	105
\$SDI32 and \$SDO32	109
\$GI and \$GO	109
\$FDIN and \$FDOUT	113
\$HDIN	117
\$TIMER, \$TIMER_S	118
\$PROG_TIMER_xx	118
Other Ports	118
\$BIT	119
\$WORD	119
System State After Restart	119
Cold Start	119
Power Failure	120
6. EXECUTION CONTROL	121
Flow Control	121
IF Statement	122
SELECT Statement	123
FOR Statement	125
WHILE Statement	126
REPEAT Statement	127
GOTO Statement	128
Program Control	129
PROG_STATE Built-In Function	130
ACTIVATE Statement	130
PAUSE Statement	131

UNPAUSE Statement	131
DEACTIVATE Statement	132
CYCLE and EXIT CYCLE Statements	132
DELAY Statement	133
WAIT FOR Statement	133
BYPASS Statement	133
Program Synchronization	134
Program Scheduling	135
7. ROUTINES	136
Procedures and Functions	137
Parameters	137
Declarations	138
Declaring a Routine	138
Procedure	139
Function	139
Parameter List	139
Constant and Variable Declarations	139
Stack Size	140
Function Return Type	141
Functions as procedures	141
RETURN Statement	141
Shared Routines	141
Passing Arguments	142
Passing By Reference	143
Passing By Value	143
Optional parameters	144
Variable arguments (Varargs)	144
Argument identifier	145
8. CONDITION HANDLERS	146
Operations	146
Defining Condition Handlers	146
FOR ARM Clause	147
NODISABLE Clause	147
ATTACH Clause	148
SCAN Clause	148
Enabling, Disabling, and Purging	149
Variable References	150
Conditions	150
RELATIONAL States	151
BOOLEAN State	151
DIGITAL PORT States	152
DIGITAL PORT Events	152
SYSTEM Events	152
USER Events	156

ERROR Events	157
PROGRAM Events	157
Event on Cancellation of a Suspendable Statement	158
MOTION Events	159
Actions	161
ASSIGNMENT Action	161
INCREMENT and DECREMENT Action	162
BUILT-IN Actions	162
SEMAPHORE Action	162
MOTION and ARM Actions	163
ALARM Actions	163
PROGRAM Actions	163
CONDITION HANDLER Actions	164
DETACH Action	164
PULSE Action	164
HOLD Action	164
SIGNAL EVENT Action	164
ROUTINE CALL Action	165
Execution Order	165
9. COMMUNICATION	167
Devices	167
WINDOW Devices	167
FILE Devices	168
PIPE Devices	169
PDL2 Devices	169
Sample Program for PDL2 Serial device use	171
NULL Device	172
Logical Unit Numbers	172
E-Mail functionality	172
Configuration of SMTP and POP3 clients	173
Sending/receiving e-mails on C5G Controller	173
“email” program	174
Sending PDL2 commands via e-mail	178
10. STATEMENTS LIST	179
Introduction	179
ACTIVATE Statement	180
ATTACH Statement	181
BEGIN Statement	183
BYPASS Statement	183
CALL Statement	184
CALLS Statement	185
CANCEL Statement	185
CLOSE FILE Statement	187

CLOSE HAND Statement	187
CONDITION Statement	187
CONST Statement	189
CYCLE Statement	190
DEACTIVATE Statement	190
DECODE Statement	191
DELAY Statement	192
DETACH Statement	193
DISABLE CONDITION Statement	194
DISABLE INTERRUPT Statement	195
ENABLE CONDITION Statement	196
ENCODE Statement	196
END Statement	197
EXIT CYCLE Statement	198
FOR Statement	198
GOTO Statement	199
HOLD Statement	200
IF Statement	200
IMPORT Statement	201
LOCK Statement	202
MOVE Statement	202
MOVE ALONG Statement	204
OPEN FILE Statement	205
WITH Clause	206
OPEN HAND Statement	207
PAUSE Statement	208
PROGRAM Statement	208
PULSE Statement	210
PURGE CONDITION Statement	210
READ Statement	211
Format Specifiers	212
Power Failure Recovery	213
RELAX HAND Statement	213
REPEAT Statement	214
RESUME Statement	214
RETURN Statement	215
ROUTINE Statement	215

SELECT Statement	216
SIGNAL Statement	217
TYPE Statement	218
UNLOCK Statement.	219
UNPAUSE Statement	220
VAR Statement	220
WAIT Statement.	222
WAIT FOR Statement	222
WHILE Statement	223
WRITE Statement	223
Output Buffer Flushing	224
Format Specifiers	224
Power Failure Recovery	225
11. BUILT-IN ROUTINES LIST	227
ABS Built-In Function.	232
ACOS Built-In Function	232
ACT_POST Built-in Procedure	232
ARG_COUNT Built-In Function	232
ARG_GET_VALUE Built-in Procedure	233
ARG_INFO Built-In Function	235
ARG_SET_VALUE Built-in Procedure.	236
ARM_COLL_THRS Built-In Procedure	236
ARM_COOP Built-In Procedure.	237
ARM_GET_NODE Built-In Function	237
ARM_JNTP Built-In Function	238
ARM_MCOOP Built-In Procedure	238
ARM_MOVE Built-in Procedure.	239
ARM_MOVE_ATVEL Built-in Procedure	240
ARM_NUM Built-In Function	241
ARM_PALLETIZING Built-In Procedure	241
ARM_POS Built-In Function	242
ARM_SET_NODE Built-In Procedure	243
ARM_SOFT Built-In Procedure	243
ARM_XTND Built-In Function	244
ARRAY_DIM1 Built-In Function	244
ARRAY_DIM2 Built-In Function	245

ASIN Built-In Function	245
ATAN2 Built-In Function	246
AUX_COOP Built-In Procedure	246
AUX_DRIVES Built-In Procedure	247
AUX_SET Built-In Procedure	247
BIT_ASSIGN Built-In Procedure	248
BIT_CLEAR Built-In Procedure	249
BIT_FLIP Built-In Function	249
BIT_SET Built-In Procedure	250
BIT_TEST Built-In Function	250
CHR Built-In Procedure	251
CLOCK Built-In Function	251
COND_ENABLED Built-In Function	252
COND_ENBL_ALL Built-In Procedure	252
CONV_TRACK Built-In Procedure	253
COS Built-In Function	254
DATE Built-In Function	254
DIR_GET Built-In Function	255
DIR_SET Built-In Procedure	255
DRIVEON_DSBL Built-In Procedure	256
DV_CNTRL Built-In Procedure	256
DV_STATE Built-In Function	262
EOF Built-In Function	263
ERR_POST Built-In Procedure	263
ERR_STR Built-In Function	266
ERR_TRAP Built-In Function	267
ERR_TRAP_OFF Built-In Procedure	267
ERR_TRAP_ON Built-In Procedure	268
EXP Built-In Function	269
FL_BYTES_LEFT Built-In Function	269
FL_GET_POS Built-In Function	269
FL_SET_POS Built-In Procedure	270
FL_STATE Built-In Function	271
FLOW_MOD_ON Built-In Procedure	272
FLOW_MOD_OFF Built-In Procedure	272
HDIN_READ Built-In Procedure	273

HDIN_SET Built-In Procedure	273
IP_TO_STR Built-in Function	276
IR_SET Built-In Procedure	276
IR_SET_DIG Built-In Procedure	277
IR_SWITCH Built-In Procedure	278
IS_FLY Built-In Function	278
JNT Built-In Procedure	279
JNT_SET_TAR Built-In Procedure	280
JNTP_TO_POS Built-In Procedure	281
KEY_LOCK Built-In Procedure	281
LN Built-In Function	282
MEM_SPACE Built-In Procedure	282
NODE_APP Built-In Procedure	283
NODE_DEL Built-In Procedure	283
NODE_GET_NAME Built-In Procedure	283
NODE_INS Built-In Procedure	284
NODE_SET_NAME Built-In Procedure	284
ON_JNT_SET Built-In Procedure	285
ON_JNT_SET_DIG Built-In Procedure	287
ON_POS Built-In Procedure	288
ON_POS_SET Built-In Procedure	289
ON_POS_SET_DIG Built-In Procedure	290
ON_TRAJ_SET Built-In Procedure	290
ON_TRAJ_SET_DIG Built-In Procedure	291
ORD Built-In Function	291
PATH_GET_NODE Built-In Procedure	292
PATH_LEN Built-In Function	292
POS Built-In Function	293
POS_COMP_IDL Built-In Procedure	293
POS_CORRECTION Built-In Procedure	294
POS_FRAME Built-In Function	294
POS_GET_APPR Built-In Function	295
POS_GET_CNFG Built-In Function	295
POS_GET_LOC Built-In Function	295
POS_GET_NORM Built-In Function	296
POS_GET_ORNT Built-In Function	296

POS_GET_RPY Built-In Procedure	296
POS_IDL_COMP Built-In Procedure	297
POS_IN_RANGE Built-In Procedure	297
POS_INV Built-In Function	298
POS_MIR Built-In Function	299
POS_SET_APPR Built-In Procedure	299
POS_SET_CNFG Built-In Procedure	300
POS_SET_LOC Built-In Procedure	300
POS_SET_NORM Built-In Procedure	300
POS_SET_ORNT Built-In Procedure	301
POS_SET_RPY Built-In Procedure	301
POS_SHIFT Built-In Procedure	301
POS_TO_JNTP Built-In Procedure	302
POS_XTRT Built-In Procedure	302
PROG_STATE Built-In Function	303
RANDOM Built-in Function	304
ROUND Built-In Function	305
SCRN_ADD Built-In Procedure	305
SCRN_CLEAR Built-In Procedure	306
SCRN_CREATE Built-In Function	306
SCRN_DEL Built-In Procedure	307
SCRN_GET Built-In Function	307
SCRN_REMOVE Built-In Procedure	308
SCRN_SET Built-In Procedure	308
SENSOR_GET_DATA Built-In Procedure	309
SENSOR_GET_OFST Built-In Procedure	309
SENSOR_SET_DATA Built-In Procedure	310
SENSOR_SET_OFST Built-In Procedure	310
SENSOR_TRK Built-In Procedure	311
SIN Built-In Function	312
SQRT Built-In Function	312
STANDBY Built-In Procedure	312
STR_CAT Built-In Function	313
STR_CODING Built-In Function	313
STR_CONVERT Built-In Function	314
STR_DEL Built-In Procedure	314

STR_EDIT Built-In Procedure	315
STR_GET_INT Built-In Function	315
STR_GET_REAL Built-In Function	315
STR_INS Built-In Procedure	316
STR_LEN Built-In Function	316
STR_LOC Built-In Function	317
STR_OVS Built-In Procedure	318
STR_SET_INT Built-In Procedure	319
STR_SET_REAL Built-In Procedure	319
STR_TO_IP Built-In Function	320
STR_TOKENS Built-In Function	320
STR_XTRT Built-In Procedure	321
SYS_ADJUST Built-In Procedure	321
SYS_CALL Built-In Procedure	322
SYS_SETUP Built-In Procedure	323
TABLE_ADD Built-In Procedure	323
TABLE_DEL Built-In Procedure	324
TAN Built-In Function	325
TREE_ADD Built-In Procedure	325
TREE_CLONE Built-In Procedure	326
TREE_CREATE Built-In Function	326
TREE_DEL Built-In Procedure	327
TREE_LOAD Built-In Procedure	327
TREE_NODE_INFO Built-In Procedure	328
TREE_NODE_CHILD Built-In Procedure	328
TREE_REMOVE Built-In Procedure	329
TREE_SAVE Built-In Procedure	329
TRUNC Built-In Function	329
VAR_CLONE Built-in Procedure	330
VAR_CREATE Built-In Procedure	330
VAR_INFO Built-In Function	331
VAR_LINK Built-In Procedure	332
VAR_LINKS Built-In Procedure	333
VAR_LINKSS Built-In Procedure	334
VAR_LINK_INFO Built-In Procedure	335
VAR_UNINIT Built-In Function	335

VAR_UNLINK Built-In Procedure	336
VAR_UNLINK_ALL Built-In Procedure	336
VEC Built-In Function	336
VOL_SPACE Built-In Procedure	337
WIN_ATTR Built-In Procedure	337
WIN_CLEAR Built-In Procedure	338
WIN_COLOR Built-In Procedure	339
WIN_CREATE Built-In Procedure	340
WIN_DEL Built-In Procedure	341
WIN_DISPLAY Built-In Procedure	341
WIN_GET_CRSR Built-In Procedure	342
WIN_LINE Built-In Function	343
WIN_LOAD Built-In Procedure	344
WIN_POPUP Built-In Procedure	345
WIN_REMOVE Built-In Procedure	346
WIN_SAVE Built-In Procedure	346
WIN_SEL Built-In Procedure	348
WIN_SET_CRSR Built-In Procedure	348
WIN_SIZE Built-In Procedure	349
WIN_STATE Built-In Function	350
12. PREDEFINED VARIABLES LIST	352
Memory Category	352
Load Category	352
Minor Category	353
Data Type	353
Attributes	353
Limits	354
S/W Version	354
Unparsed	354
Predefined Variables groups	354
PORT System Variables	355
PROGRAM STACK System Variables	355
PROGRAM LOADED System Variables	356
ARM_DATA System Variables	357
PROG_ARM_DATA System Variables	360
CRNT_DATA System Variables	360
INTERFERENCE REGIONs System Variables	361
WEAVE_TBL System Variables	361

ON_POS_TBL System Variables	362
TRK_TBL System Variables	362
WITH MOVE System Variables	362
WITH MOVE ALONG System Variables	363
WITH OPEN FILE System Variables	365
PATH NODE FIELD System Variables	365
IO_DEV System Variables	366
IO_STS System Variables	366
LOOP_DATA System Variables	366
MISCELLANEOUS System Variables	367
 Alphabetical Listing	370
\$A_ALONG_1D: Internal arm data	382
\$A_ALONG_2D: Internal arm data	382
\$A_AREAL_1D: Internal arm data	383
\$A_AREAL_2D: Internal arm data	383
\$AIN: Analog input	383
\$AOUT: Analog output	384
\$APPL_ID: Application Identifier	384
\$APPL_NAME: Application Identifiers	384
\$APPL_OPTIONS: Application Options	385
\$ARM_ACC_OVR: Arm acceleration override	385
\$ARM_DATA: Robot arm data	385
\$ARM_DEC_OVR: Arm deceleration override	386
\$ARM_DISB: Arm disable flags	386
\$ARM_ID: Arm identifier	386
\$ARM_LINKED: Enable/disable arm coupling	387
\$ARM_OVR: Arm override	387
\$ARM_SENSITIVITY: Arm collision sensitivity	387
\$ARM_SIMU: Arm simulate flag	388
\$ARM_SPACE: current Arm Space	388
\$ARM_SPD_OVR: Arm speed override	389
\$ARM_USED: Program use of arms	389
\$ARM_VEL: Arm velocity	389
\$AUX_BASE: Auxiliary base for a positioner of an arm	390
\$AUX_KEY: Teach Pendant AUX-A and AUX-B keys mapping	390
\$AUX_MASK: Auxiliary arm mask	390
\$AUX_OFST: Auxiliary axes offsets	391
\$AUX_TYPE: Positioner type	391
\$AX_CNVRSN: Axes conversion	391

\$AX_INF: Axes inference	392
\$AX_LEN: Axes lengths	393
\$AX_OFST: Axes offsets	393
\$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature	393
\$AX_PURSUIT_ENBL: Axes Pursuit enabling flag	394
\$AX_PURSUIT_LINKED: Axes Pursuit linked	394
\$B_ALONG_1D_NS: Internal board data	395
\$B_ASTR_1D_NS: Board string data	395
\$BACKUP_SET: Default devices	395
\$BASE: Base of arm	396
\$BIT: BIT data	396
\$BOOTLINES: Bootline read-only	397
\$BREG: Boolean registers - saved	397
\$BREG_NS: Boolean registers - not saved	397
\$B_NVRAM: NVRAM data of the board	398
\$C_ALONG_1D: Internal current arm data	398
\$C_AREAL_1D: Internal current arm data	398
\$C_AREAL_2D: Internal current arm data	398
\$CAL_DATA: Calibration data	399
\$CAL_SYS: System calibration position	399
\$CAL_USER: User calibration position	400
\$CAUX_POS: Cartesian positioner position	400
\$CIO_TREEID: Tree identifier for the configure I/O tree	400
\$CNFG_CARE: Configuration care	401
\$CNTRL_CNFG: Controller configuration mode	401
\$CNTRL_DST: Controller Day Light Saving	402
\$CNTRL_INIT: Controller initialization mode	402
\$CNTRL_OPTIONS: Controller Options	403
\$CNTRL_TZ: Controller Time Zone	404
\$COLL_EFFECT: Collision Effect on the arm status	404
\$COLL_ENBL: Collision enabling flag	405
\$COLL_SOFT_PER: Collision compliance percentage	405
\$COLL_TYPE: Type of collision	405
\$COND_MASK: PATH segment condition mask	406
\$COND_MASK_BACK: PATH segment condition mask in backwards	407
\$CONV_ACT: Conveyor Act	407

\$CONV_BEND_ANGLE: Conveyor Bend Angle	408
\$CONV_DIST: Conveyor shift in micron	408
\$CONV_ENC: Conveyor Encoder Position	408
\$CONV_SHIFT: Conveyor shift in mm	409
\$CONV_SPD: Conveyor Speed	409
\$CRC_IMPORT: Directory for IMPORT.	409
\$CRC_RULES: C5G Save & Load rules	410
\$CRNT_DATA: Current Arm data	411
\$CUSTOM_CNTRL_ID: Identifier for the Controller	411
\$CUSTOM_ID: Identifier for the arm.	411
\$CYCLE: Program cycle count	411
\$DB_MSG: Message Database Identifier	412
\$DELTA_POS: offsets on final position in the specified (\$DELTA_POS_IN) frame	412
\$DELTA_POS_IN: X, Y, Z offsets on final position	413
\$DEPEND_DIRS: Dependancy path	413
\$DFT_ARM: Default arm	413
\$DFT_DV: Default devices	414
\$DFT_LUN: Default LUN number	414
\$DFT_SPD: Default devices speed	415
\$DIN: Digital input	415
\$DNS_DOMAIN: DNS Domain	415
\$DNS_ORDER: DNS Order.	416
\$DNS_SERVERS: DNS Servers.	416
\$DOUT: Digital output	416
\$DV_STS: the status of DV_CNTRL calls	417
\$DV_TOUT: Timeout for asynchronous DV_CNTRL calls	417
\$DYN_COLL_FILTER: Dynamic Collision Filter	417
\$DYN_DELAY: Dynamic model delay	417
\$DYN_FILTER2: Dynamic Filter for dynamic model	418
\$DYN_GAIN: Dynamic gain in inertia and viscous friction.	418
\$DYN_MODEL: Dynamic Model	419
\$DYN_WRIST: Dynamic Wrist.	419
\$DYN_WristQs: Dynamic Theta	419
\$EMAIL_INT: Email integer configuration	419
\$EMAIL_STR: Email string configuration.	420
\$ERROR: Last PDL2 Program Error	420

\$ERROR_DATA: Last PDL2 Program Error Data	421
\$EXE_HELP: Help on Execute command	421
\$FDIN: Functional digital input	421
\$FDOUT: Functional digital output	422
\$FEED_VOLTAGE: Feed Voltage	422
\$FL_ADLM: Array of delimiters	422
\$FL_ARM_CNFG: Arm configuration file name	423
\$FL_AX_CNFG: Axes configuration file name	423
\$FL_BINARY: Text or character mode	423
\$FL_CNFG: Configuration file name	424
\$FL_COMP: Compensation file name	424
\$FL_DLMT: Delimiter specification	424
\$FL_ECHO: Echo characters	425
\$FL_NUM_CHARS: Number of chars to be read	425
\$FL_PASSALL: Pass all characters	425
\$FL_RANDOM: Random file access	426
\$FL_RDFLUSH: Flush on reading	426
\$FL_STS: Status of last file operation	426
\$FL_SWAP: Low or high byte first	427
\$FLOW_TBL: Flow modulation algorithm table data	427
\$FLY_DBUG: Cartesian Fly Debug	427
\$FLY_DIST: Distance in fly motion	428
\$FLY_PER: Percentage of fly motion	428
\$FLY_TRAJ: Type of control on cartesian fly	428
\$FLY_TYPE: Type of fly motion	429
\$FMI: Flexible Multiple Analog/Digital Inputs	429
\$FMO: Flexible Multiple Analog/Digital Outputs	429
\$FOLL_ERR: Following error	430
\$FUI_DIRS: Installation path	430
\$FW_ARM: Arm under flow modulation algorithm	430
\$FW_AXIS: Axis under flow modulation algorithm	431
\$FW_CNVRSN: Conversion factor in case of Flow modulation algorithm	431
\$FW_ENBL Flow modulation algorithm enabling indicator	431
\$FW_FLOW_LIM Flow modulation algorithm flow limit	432
\$FW_SPD_LIM Flow modulation algorithm speed limits	432
\$FW_START Delay in flow modulation algorithm application after start	432

\$FW_VAR: flag defining the variable to be considered when flow modulate is used	433
\$GEN_OVR: General override.	433
\$GI: General System Input	433
\$GO: General System Output	434
\$GUN: Electrical welding gun	434
\$HAND_TYPE: Type of hand	434
\$HDIN: High speed digital input.	435
\$HDIN_SUSP: temporarily disables the use of \$HDIN	435
\$HLD_DEC_PER: Hold deceleration percentage	435
\$HOME: Arm home position	436
\$ID_ADRS: Address on the Fieldbus net.	436
\$ID_APPL: Application Code.	436
\$ID_DATA: Fieldbus specific data.	437
\$ID_ERR: error Code.	437
\$ID_IDX: Device index.	437
\$ID_ISIZE: Input bytes quantity	437
\$ID_ISMST: this Device is a Master (or Controller)	438
\$ID_MASK: miscellaneous mask of bits	438
\$ID_NAME: Device name	438
\$ID_NET: Fieldbus type.	439
\$ID_NOT_ACTIVE: not active Device at net boot time	439
\$ID_OSIZE: output bytes quantity	439
\$ID_STS: Device status.	440
\$ID_TYPE: configuration status.	440
\$IN: IN digital	440
\$IO_DEV: Input/Output Device Table	440
\$IO_STS: Input/Output point StatusTable	441
\$IPERIOD: Interpolator period.	442
\$IREG: Integer register - saved	442
\$IREG_NS: Integer registers - not saved	442
\$IR_TBL: Interference Region table data	443
\$IR_ARM: Interference Region Arm	443
\$IR_CNFG: Interference Region Configuration.	444
\$IR_CON_LOGIC: Interference Region Consent Logic.	444
\$IR_ENBL: Interference Region enabled flag	444
\$IR_IN_OUT: Interference Region location flag	445

\$IR_JNT_N: Interference Region negative joint	445
\$IR_JNT_P: Interference Region positive joint	445
\$IR_PBIT: Interference Region port bit	446
\$IR_PBIT_RC: Interference Region port bit for reservation and consent	446
\$IR_PERMANENT: Interference Region permanent.....	446
\$IR_PIDX: Interference Region port index.....	446
\$IR_PIDX_RC: Interference Region port index for reservation and consent.....	447
\$IR_POS1: Interference Region position.....	447
\$IR_POS2: Interference Region position.....	447
\$IR_POS3: Interference Region position.....	448
\$IR_POS4: Interference Region position.....	448
\$IR_PTYPE: Interference Region port type	448
\$IR_PTYPE_RC: Interference Region port type for reservation and consent	448
\$IR_REACHED: Interference Region position reached flag	449
\$IR_RES_LOGIC: Interference Region reservation logic	449
\$IR_SHAPE: Interference Region Shape	449
\$IR_TYPE: Interference Region type.....	450
\$IR_UFRAME: Interference Region Uframe	450
\$IS_DEV: associated Device	450
\$IS_HELP_INT: User Help code	451
\$IS_HELP_STR: User Help string	451
\$IS_NAME: I/O point name	451
\$IS_PORT_CODE: Port type.....	452
\$IS_PORT_INDEX: Port type index	452
\$IS_PRIV: privileged Port flag	452
\$IS_RTN: retentive output flag.....	453
\$IS_SIZE: Port dimension	453
\$IS_STS: Port status	453
\$JERK: Jerk control values	454
\$JL_TABLE: Internal Arm data	454
\$JNT_LIMIT_AREA: Joint limits of the work area	454
\$JNT_MASK: Joint arm mask	455
\$JNT_MTURN: Check joint Multi-turn	455
\$JNT_OVR: joint override	455
\$JNT_SM: joint trajectory planning indicator	456
\$JOG_INCR_DIST: Increment Jog distance	456

\$JOG_INCR_ENBL: Jog incremental motion	456
\$JOG_INCR_ROT: Rotational jog increment	457
\$JOG_SPD_OVR: Jog speed override	457
\$JPAD_DIST: Distance between user and Jpad.	457
\$JPAD_ORNT: Teach Pendant Angle setting	458
\$JPAD_TYPE: TP Jpad modality rotational or translational	458
\$JREG: Jointpos registers - saved	458
\$JREG_NS: Jointpos register - not saved	459
\$L_ALONG_1D: Internal Loop data.	459
\$L_ALONG_2D: Internal Loop data.	459
\$L_AREAL_1D: Internal Loop data	459
\$L_AREAL_2D: Internal Loop data	460
\$LAD_OVR: TP Linear Acc/Dec Override for FLY CART	460
\$LAST_JNT_AXIS: Number of the last joint of the arm.	460
\$LATCH_CNFG: Latched alarm configuration setting.	461
\$LIN_ACC_LIM: Linear acceleration limit	461
\$LIN_DEC_LIM: Linear deceleration limit	461
\$LIN_SPD: Linear speed	462
\$LIN_SPD_LIM: Linear speed limit	462
\$LIN_SPD_RT: Run-time Linear speed.	462
\$LIN_SPD_RT_OVR: Run-time Linear speed override.	463
\$LOG_TO_CHANNEL: Logical to channel relationship.	463
\$LOG_TO_DRV: Logical to physical drives relationship	463
\$LOOP_DATA: Loop data	464
\$MAIN_JNTP: PATH node main jointpos destination	464
\$MAIN_POS: PATH node main position destination	464
\$MAIN_XTND: PATH node main xtndpos destination.	465
\$MAN_SCALE: Manual scale factor	465
\$MDM_INT: Modem Configuration	465
\$MDM_STR: Modem Configuration.	466
\$MOD_ACC_DEC: Modulation of acceleration and deceleration	466
\$MOD_MASK: Joint mod mask	466
\$MOVE_STATE: Move state	467
\$MOVE_TYPE: Type of motion	467
\$MTR_ACC_TIME: Motor acceleration time	467
\$MTR_CURR: Motor current	468

\$MTR_DEC_TIME: Motor deceleration time	468
\$MTR_SPD_LIM: Motor speed limit	468
\$NET_B: Ethernet Boot Setup	468
\$NET_B_DIR: Ethernet Boot Setup Directory	469
\$NET_C_CNGF: Ethernet Client Setting Modes	469
\$NET_C_DIR: Ethernet Client Setup Default Directory	470
\$NET_C_HOST: Ethernet Client Setup Remote Host	470
\$NET_C_PASS: Ethernet Client Setup Password	470
\$NET_C_USER: Ethernet Client Setup Login Name	471
\$NET_HOSTNAME: Ethernet network hostnames	471
\$NET_I_INT: Ethernet Network Information (integers)	471
\$NET_I_STR: Ethernet Network Information (strings)	472
\$NET_L: Ethernet Local Setup	472
\$NET_MOUNT: Ethernet network mount	473
\$NET_Q_STR: Ethernet Remote Interface Information	473
\$NET_R_STR: Ethernet Remote Interface Setup	474
\$NET_S_INT: Ethernet Network Server Setup	474
\$NET_T_HOST: Ethernet Network Time Protocol Host	475
\$NET_T_INT: Ethernet Network Timer	475
\$NOLOG_ERROR: Exclude messages from logging	475
\$NUM_ALOG_FILES: Number of action log files	476
\$NUM_ARMS: Number of arms	476
\$NUM_AUX_AXES: Number of auxiliary axes	476
\$NUM_DBs: Number of allowed Databases	476
\$NUM_DEVICES: Number of Devices	477
\$NUM_IO_DEV: Number of Input/Output Devices	477
\$NUM_IO_STS: Number of Input/Output points	477
\$NUM_JNT_AXES: Number of joint axes	478
\$NUM_LUNS: Number of LUNs	478
\$NUM_MB: Number of motion buffers	478
\$NUM_MB_AHEAD: Number of motion buffers ahead	478
\$NUM_PROGS: Number of active programs	479
\$NUM_PROG_TIMERS: Number of program timers (\$PROG_TIMER_xxx)	479
\$NUM_SCRNS: Number of screens	479
\$NUM_TIMERS: Number of timers	480
\$NUM TREES: Number of trees	480

\$NUM_VP2_SCRNS: Number of Visual PDL2 screens	480
\$NUM_WEAVES: Number of weaves (WEAVE_TBL)	481
\$ODO_METER: average TCP space	481
\$ON_POS_TBL: ON POS table data.	481
\$OP_JNT: On Pos jointpos	482
\$OP_JNT_MASK: On Pos Joint Mask.	482
\$OP_POS: On Pos position.	482
\$OP_REACHED: On Pos position reached flag	483
\$OP_TOL_DIST: On Pos-Jnt Tolerance distance	483
\$OP_TOL_ORNT: On Pos-Jnt Tolerance Orientation	483
\$OP_TOOL: The On Pos Tool.	484
\$OP_TOOL_DSBL: On Pos tool disable flag.	484
\$OP_TOOL_RMT: On Pos Remote tool flag.	484
\$OP_UFRAME: The On Pos Uframe.	485
\$ORNT_TYPE: Type of orientation	485
\$OT_COARSE: On Trajectory indicator	485
\$OT_JNT: On Trajectory joint position.	485
\$OT_POS: On Trajectory position	486
\$OT_TOL_DIST: On Trajectory Tolerance distance	486
\$OT_TOL_ORNT: On Trajectory Orientation	486
\$OT_TOOL: On Trajectory TOOL position	487
\$OT_TOOL_RMT: On Trajectory remote tool flag.	487
\$OT_UFRAME: On Trajectory User frame	487
\$OT_UNINIT: On Trajectory position uninit flag	488
\$OUT: OUT digital	488
\$PAR: Nodal motion variable.	488
\$PGOV_ACCURACY: required accuracy in cartesian motions.	490
\$PGOV_MAX_SPD_REDUCTION: Maximum speed scale factor	490
\$PGOV_ORNT_PER: percentage of orientation	490
\$POS_LIMIT_AREA: Cartesian limits of work area.	491
\$PREG: Position registers - saved	491
\$PREG_NS: Position registers - not saved	491
\$PROG_ACC_OVR: Program acceleration override.	492
\$PROG_ARG: Program's activation argument	492
\$PROG_ARM: Arm of program	492
\$PROG_ARM_DATA: Arm-related Data - Program specific	493

\$PROG_BREG: Boolean Registers - Program specific	493
\$PROG_CNFG: Program configuration	493
\$PROG_CONDS: Defined conditions of a program	494
\$PROG_DEC_OVR: Program deceleration override.	495
\$PROG_IREG: Integer Registers - Program specific	495
\$PROG_LINE: executing program line	495
\$PROG_LUN: program specific default LUN.	496
\$PROG_NAME: Executing program name	496
\$PROG_OWNER: Program Owner of executing line	496
\$PROG_RES: Program's execution result	497
\$PROG_RREG: Real Registers - Program specific	497
\$PROG_SPD_OVR: Program speed override.	497
\$PROG_SREG: String Registers - Program specific.	497
\$PROG_TIMER_O: Program timer - owning context specific (in ms)	498
\$PROG_TIMER_OS: Program timer - owning context specific (in seconds)	498
\$PROG_TIMER_X: Program timer - execution context specific (in ms)	499
\$PROG_TIMER_XS: Program timer - execution context specific (in seconds)	499
\$PROP_AUTHOR: last Author who saved the file.	499
\$PROP_DATE: date and time when the program was last saved.	500
\$PROP_FILE: property information for loaded file.	500
\$PROP_HELP: the help the user wants	501
\$PROP_HOST: Controller ID or PC user domain, upon which the file was last saved	501
\$PROP_REVISION: user defined string representing the version	501
\$PROP_TITLE: the title defined by the user for the file	502
\$PROP_UML: user modify level	502
\$PROP_UVL: user view level	503
\$PROP_VERSION: version upon which it was built	503
\$PWR_RCVR: Power failure recovery mode.	503
\$RAD_IDL_QUO: Radiant ideal quote.	504
\$RAD_OVR: TP Rotational Acc/Dec Override for FLY CART	504
\$RAD_TARG: Radiant target.	504
\$RAD_VEL: Radiant velocity	505
\$RBT_CNFG: Robot board configuration	505
\$RB_FAMILY: Family of the robot arm	506
\$RB_MODEL: Model of the robot arm	506
\$RB_NAME: Name of the robot arm	506

\$RB_STATE: State of the robot arm	507
\$RB_VARIANT: Variant of the robot arm	507
\$RCVR_DIST: Distance from the recovery position	507
\$RCVR_LOCK: Change arm state after recovery	508
\$RCVR_TYPE: Type of motion recovery	508
\$READ_TOUT: Timeout on a READ	509
\$REC_SETUP: RECord key setup	509
\$REMOTE: Functionality of the key in remote	509
\$REM_I_STR: Remote connections Information	510
\$REM_TUNE: Internal remote connection tuning parameters	510
\$RESTART: Restart Program	510
\$RESTART_MODE: Restart mode	511
\$RESTORE_SET: Default devices	511
\$ROT_ACC_LIM: Rotational acceleration limit	512
\$ROT_DEC_LIM: Rotational deceleration limit	512
\$ROT_SPD: Rotational speed	512
\$ROT_SPD_LIM: Rotational speed limit	513
\$RREG: Real registers - saved	513
\$RREG_NS: Real registers - not saved	513
\$SAFE_ENBL: Safe speed enabled	513
\$SAFE_SPD: User Velocity for testing safety speed	514
\$SDI: System digital input	514
\$SDO: System digital output	514
\$SEG_ADV: PATH segment advance	515
\$SEG_COND: PATH segment condition	515
\$SEG_DATA: PATH segment data	515
\$SEG_FLY: PATH segment fly or not	516
\$SEG_FLY_DIST: Parameter in segment fly motion	516
\$SEG_FLY_PER: PATH segment fly percentage	516
\$SEG_FLY_TRAJ: Type of fly control	517
\$SEG_FLY_TYPE: PATH segment fly type	517
\$SEG_OVR: PATH segment override	517
\$SEG_REF_IDX: PATH segment reference index	518
\$SEG_STRESS_PER: Percentage of stress required in fly	519
\$SEG_TERM_TYPE: PATH segment termination type	519
\$SEG_TOL: PATH segment tolerance	519

\$SEG_TOOL_IDX: PATH segment tool index.....	520
\$SEG_WAIT: PATH segment WAIT	520
\$SENSOR_CNVRSN: Sensor Conversion Factors.....	521
\$SENSOR_ENBL: Sensor Enable.....	522
\$SENSOR_GAIN: Sensor Gains	522
\$SENSOR_OFST_LIM: Sensor Offset Limits	522
\$SENSOR_TIME: Sensor Time.....	523
\$SENSOR_TYPE: Sensor Type	523
\$SERIAL_NUM: Serial Number of the board.....	524
\$SFRAME: Sensor frame of an arm	524
\$SING_CARE: Singularity care	524
\$SM4_SAT_ADD_SCALE: Additional threshold for the SmartMove saturation	525
\$SM4C_SAT_VEL_SCALE: thresholds for the joint speed saturation in Cartesian SmartMove..	525
\$SM4C_STRESS_PER: Maximum Stress allowed in Cartesian SmartMove	525
\$SPD_OPT: Type of speed control	526
\$SREG: String registers - saved	526
\$SREG_NS: String registers - not saved.....	526
\$STARTUP: Startup program	527
\$STARTUP_USER: Configuration file name	527
\$STRESS_PER: Stress percentage in cartesian fly	527
\$STRK_END_N: User negative stroke end	528
\$STRK_END_P: User positive stroke end	528
\$STRK_END_SYS_N: System stroke ends	528
\$STRK_END_SYS_P: System stroke ends.....	528
\$SYNC_ARM: Synchronized arm of program	529
\$SYS_CALL_OUT: Output lun for SYS_CALL	529
\$SYS_CALL_STS: Status of last SYS_CALL	529
\$SYS_CALL_TOUT: Timeout for SYS_CALL	530
\$SYS_ERROR: Last system error.....	530
\$SYS_ID: Robot System identifier.....	530
\$SYS_OPTIONS: System options.....	531
\$SYS_PARAMS: Robot system identifier	531
\$SYS_RESTART: System Restart Program	531
\$SYS_STATE: State of the system	531
\$TERM_TYPE: Type of motion termination.....	532
\$THRD_CEXP: Thread Condition Expression.....	533

\$THRD_ERROR: Error of each thread of execution	534
\$THRD_PARAM: Thread Parameter	534
\$TIMER: Clock timer (in ms)	535
\$TIMER_S: Clock timer (in seconds).....	535
\$TOL_ABST: Tolerance anti-bounce time	535
\$TOL_COARSE: Tolerance coarse.....	536
\$TOL_FINE: Tolerance fine.....	536
\$TOL_JNT_COARSE: Tolerance for joints	536
\$TOL_JNT_FINE: Tolerance for joints.....	537
\$TOL_TOUT: Tolerance timeout.....	537
\$TOOL: Tool of arm.....	537
\$TOOL_CNTR: Tool center of mass of the tool.....	538
\$TOOL_INERTIA: Tool Inertia.....	538
\$TOOL_MASS: Mass of the tool	538
\$TOOL_RMT: Fixed Tool.....	538
\$TOOL_XTREME: Extreme Tool of the Arm.....	539
\$TP_ARM: Teach Pendant current arm.....	539
\$TP_GEN_INCR: Incremental value for general override.....	539
\$TP_MJOG: Type of TP jog motion.....	540
\$TP_ORNT: Orientation for jog motion	540
\$TP_SYNC_ARM: Teach Pendant's synchronized arms	540
\$TRK_TBL: Tracking application table data	541
\$TT_APPL_ID: Tracking application identifier.....	541
\$TT_ARM_MASK: Tracking arm mask	542
\$TT_FRAMES: Array of POSITIONS for tracking application	542
\$TT_I_PRMS: Integer parameters for the tracking application	542
\$TT_PORT_IDX: Port Index for the tracking application	543
\$TT_PORT_TYPE: Port Type for the tracking application.....	543
\$TT_R_PRMS: Real parameters for the tracking application	543
\$TUNE: Internal tuning parameters.....	544
\$TURN_CARE: Turn care	545
\$TX_RATE: Transmission rate	545
\$UDB_FILE: Name of UDB file	546
\$UFRAAME: User frame of an arm	546
\$VERSION: Software version	546
\$VP2_SCRN_ID: Executing program VP2 Screen Identifier.....	546

\$VP2_TOUT: Timeout value for asynchronous VP2 requests.....	547
\$VP2_TUNE: Visual PDL2 tuning parameters.....	547
\$WEAVE_MODALITY: Weave modality	547
\$WEAVE_MODALITY_NOMOT: Weave modality (only for no arm motion)	548
\$WEAVE_NUM: Weave table number.....	548
\$WEAVE_NUM_NOMOT: Weave table number (only for no arm motion)	548
\$WEAVE_PHASE: Index of the Weaving Phase.....	549
\$WEAVE_TBL: Weave table data	549
\$WEAVE_TYPE: Weave type	549
\$WEAVE_TYPE_NOMOT: Weave type (only for no arm motion)	550
\$WFR_IOTOUT: Timeout on a WAIT FOR when IO simulated.....	550
\$WFR_TOUT: Timeout on a WAIT FOR	550
\$WORD: WORD data	551
\$WRITE_TOUT: Timeout on a WRITE	551
\$WV_AMPLITUDE: Weave amplitude.....	551
\$WV_AMP_PER_RIGHT/LEFT: Weave amplitude percentage	552
\$WV_CNTR_DWL: Weave center dwell	552
\$WV_DIRECTION_ANGLE: Weave angle direction	552
\$WV_END_DWL Weave end dwell	553
\$WV_LEFT_AMP: Weave left amplitude.....	553
\$WV_LEFT_DWL: Weave left dwell	553
\$WV_LENGTH_WAVE: Wave length	554
\$WV_ONE_CYCLE: Weave one cycle	554
\$WV_PLANE: Weave plane angle.....	554
\$WV_PLANE_MODALITY: Weave plane modality	555
\$WV_RADIUS: Weave radius	555
\$WV_RIGHT_AMP: Weave right amplitude	555
\$WV_RIGHT_DWL: Weave right dwell	556
\$WV_SMOOTH: Weave smooth enabled	556
\$WV_SPD_PROFILE: Weave speed profile enabled	556
\$WV_TRV_SPD: Weave transverse speed.....	557
\$WV_TRV_SPD_PHASE: Weave transverse speed phase	557
\$XREG: Xtndpos registers - saved	557
\$XREG_NS: Xtndpos registers - not saved.....	558
13. POWER FAILURE RECOVERY.....	559

14. TRANSITION FROM C4G TO C5G CONTROLLER	560
Introduction	560
How to run C4G programs on C5G Controller	560
Predefined Variables list	560
Variables which have been renamed or moved to other structures	561
Removed Predefined Variables	561
New Predefined Variables	562
DV_CTRL	562
Environment variables on the PC	562
Built-in routines and functions	562
15. APPENDIX A - CHARACTERS SET	563
Characters Table	564

PREFACE

- Symbols used in the manual
- Reference documents
- Modification History

Symbols used in the manual

The symbols for **WARNING**, **CAUTION** and **NOTES** are indicated below together with their significance.



This symbol indicates operating procedures, technical information and precautions that if ignored and/or are not performed correctly could cause injuries.



This symbol indicates operating procedures, technical information and precautions that if ignored and/or are not performed correctly could cause damage to the equipment.



This symbol indicates operating procedures, technical information and precautions that it are important to highlight.

Reference documents

This document refers to the **C5G Control Unit**.

The complete set of manuals for the **C5G** consists of:

Comau	C5G Control Unit	<ul style="list-style-type: none">– Technical Specifications– Transport and installation– Guide to integration, safeties, I/O and communications– Use of Control Unit.
-------	------------------	---

These manuals are to be integrated with the following documents:

Comau	Robot	<ul style="list-style-type: none">– Technical Specifications– Transport and installation– Maintenance
	Programming	<ul style="list-style-type: none">– PDL2 Programming Language– VP2 - Visual PDL2– Motion programming
	Applications	<ul style="list-style-type: none">– According to the required type of application.

Modification History

In Manual version 02/0710, the following modifications have been made:

- [Chap.5. - Ports](#) - added \$FMI_BIT, \$FMO_BIT, \$WORD_BIT Ports.
- [Chap.11. - BUILT-IN Routines List](#) has been updated:
 - added [IR_SET Built-In Procedure](#), [IR_SET_DIG Built-In Procedure](#), [IR_SWITCH Built-In Procedure](#) to handle Interference Regions.
- [Chap.12. - Predefined Variables List](#) has been updated:
 - added group [INTERFERENCE REGIONS System Variables](#) and corresponding \$IR_xx sysvars
 - added sysvars - \$DB_MSG, \$ID_SMST, \$IS_OPTIONS, \$NUM_DBS, \$PROP_FILE, \$WV_AMPLITUDE, \$WV_DIRECTION_ANGLE, \$WV_PLANE_MODALITY, \$WV_RADIUS
 - deleted sysvars - \$FB_CNFG, \$FB_INIT
 - modified sysvars:
 - \$IS_PRIVILEGDE is now \$IS_PRIV
 - \$CNTRL_OPTIONS - added bit 31 for Interference Regions
- [Chap.14. - Transition from C4G to C5G Controller](#) has been updated:
 - par. 14.3.1 [Variables which have been renamed or moved to other structures on page 561](#) - added \$SDI, \$SDO, \$SDI32, \$SDO32
 - par. 14.3.2 [Removed Predefined Variables on page 561](#) - added \$CIO_AIN, \$CIO_AOUT, \$CIO_CROSS, \$CIO_DIN, \$CIO_DOUT, \$CIO_FMI, \$CIO_FMO, \$CIO_GIN, \$CIO_GOUT, \$CIO_IN_APP, \$CIO_OUT_APP, \$CIO_SDIN, \$CIO_SDOOUT, \$CONV_ZERO, \$CONV_WIN, \$CONV_TYPE, \$CONV_CNFG, \$CONV_BASE, \$CONV_SPD_LIM, \$CONV_ACC_LIM, \$CONV_TBL, \$CT_TX_RATE, \$CT_JNT_MASK, \$CT_SCC, \$CT_RES, \$CT_RADIUS, \$CT_SHEET_DEPTH, \$CT_CAVE_ANGLE, \$CT_SHOULDER_RADIUS, \$CT_CAVE_WIDTH, \$CT_KNIFE_RADIUS, \$CT_DELAY, \$CT_ARMA_FILTER, \$CT_MAX_BEND_ANGLE, \$CUSTOM_ARM_ID, \$FB_CNFG, \$FB_INIT, \$FB_MA_SLVS_NAME, \$FB_MA_SLVS_PRMS, \$FB_MA_SLVS_STR, \$D_HDIN_SUSP, \$GIN, \$GOUT, \$PROG_RES.
 - par. 14.3.3 [New Predefined Variables on page 562](#) - created \$GI, \$GO
 - par. 14.6 [Built-in routines and functions on page 562](#) - when calling VP2_SCRN_AVAIL Built-in Procedure, the second parameter must now be "tp" and no longer "tp4i".

In Manual version 05/2011.6, the following modifications have been made:

- [Chap.2. - Introduction to PDL2](#) - added [Predefined Constants](#) to handle Interference Regions.
- [Chap.5. - Ports](#) - added \$GI and \$GO to handle IEAK and IESK
- [Chap.11. - BUILT-IN Routines List](#)
 - added [ARM_MCOOP Built-In Procedure](#) for multi-cooperative motion
 - modified [IR_SET Built-In Procedure](#) and [IR_SET_DIG Built-In Procedure](#) to add one more parameter which represents the meaning of the values of the output bit specified by the first three parameters of [IR_SET Built-In Procedure](#) and [IR_SET_DIG Built-In Procedure](#)
- [Chap.12. - Predefined Variables List](#)
 - [\\$IR_SHAPE: Interference Region Shape](#) predefined variable description has been modified, because the following predefined constants are now available:

IR_NOT_PRESENT, IR_PARALLELEPIPED, IR_SPHERE, IR_CYLINDER, IR_PLANE and IR_JOINT

- **\$IR_TYPE:** Interference Region type predefined variable description has been modified, because the following predefined constants are now available: IR_FORBIDDEN and IR_MONITORED.

In Manual version 06/2011.09, the following modifications have been made:

- **Chap.11. - BUILT-IN Routines List**
 - added **CONV_TRACK** Built-In Procedure for Conveyor Tracking application
 - eliminated CONV_SET_OFST Built-in procedure
- **Chap.12. - Predefined Variables List**
 - **\$TRK_TBL:** Tracking application table data and its fields description, to handle Tracking application.

In Manual version 09/2012.09, the following modifications have been made:

- **Chap.5. - Ports**
 - added a note about **\$TIMER**, **\$TIMER_S** and **\$PROG_TIMER_xx** maximum allowed values before overflow
- **Chap.12. - Predefined Variables List**
 - **\$PROG_RES:** Program's execution result "program specific" predefined variables that can be used by users instead of having to define variables.

In Manual minor version 09/2012.11, the following modifications have been made:

- **Chap.2. - Introduction to PDL2**
 - added **SCRN_VP2** predefined constant
 - added new **par. 2.1 Peculiarities of PDL2 Language** on page 42
- **Chap.11. - BUILT-IN Routines List**
 - added examples and improved descriptions for both **IR_SET** Built-In Procedure and **IR_SET_DIG** Built-In Procedure
- **Chap.12. - Predefined Variables List**
 - added description for the new **\$SAFE_SPD:** User Velocity for testing safety speed predefined variable.

1. GENERAL SAFETY PRECAUTIONS



It deals with a general specification that apply to the whole Robot System. Due to its significance, this document is referred to unreservedly in any system instruction manual.

This specification deals with the following topics:

- [Responsibilities](#)
- [Safety Precautions](#).

1.1 Responsibilities

- The system integrator is responsible for ensuring that the [Robot System \(Robot and Control System\)](#) are installed and handled in accordance with the Safety Standards in force in the country where the installation takes place. The application and use of the protection and safety devices necessary, the issuing of declarations of conformity and any CE markings of the system are the responsibility of the Integrator.
- COMAU Robotics & Service shall in no way be held liable for any accidents caused by incorrect or improper use of the [Robot System \(Robot and Control System\)](#), by tampering with circuits, components or software, or the use of spare parts that are not included in the spare parts list.
- The application of these Safety Precautions is the responsibility of the persons assigned to direct / supervise the activities indicated in the [Applicability](#) sectionally are to make sure that the [Authorised Personnel](#) is aware of and scrupulously follow the precautions contained in this document as well as the Safety Standards in addition to the Safety Standards in force in the country in which it is installed.
- The non-observance of the Safety Standards could cause injuries to the operators and damage the [Robot System \(Robot and Control System\)](#).



The installation shall be made by qualified installation Personnel and should conform to all national and local codes.

1.2 Safety Precautions

1.2.1 Purpose

These safety precautions are aimed to define the behaviour and rules to be observed when performing the activities listed in the [Applicability](#) section.

1.2.2 Definitions

Robot System (Robot and Control System)

The Robot System is a functional unit consisting of Robot, Control Unit, Programming terminal and possible options.

Protected Area

The protected area is the zone confined by the safety barriers and to be used for the installation and operation of the robot

Authorised Personnel

Authorised personnel defines the group of persons who have been trained and assigned to carry out the activities listed in the [Applicability](#) section.

Assigned Personnel

The persons assigned to direct or supervise the activities of the workers referred to in the paragraph above.

Installation and Putting into Service

The installation is intended as the mechanical, electrical and software integration of the Robot and Control System in any environment that requires controlled movement of robot axes, in compliance with the safety requirements of the country where the system is installed.

Programming Mode

Operating mode under the control of the operator, that excludes automatic operation and allows the following activities: manual handling of robot axes and programming of work cycles at low speed, programmed cycle testing at low speed and, when allowed, at the working speed.

Auto / Remote Automatic Mode

Operating mode in which the robot autonomously executes the programmed cycle at the work speed, with the operators outside the protected area, with the safety barriers closed and the safety circuit activated, with local (located outside the protected area) or remote start/stop.

Maintenance and Repairs

Maintenance and repairs are activities that involve periodical checking and / or replacement (mechanical, electrical, software) of Robot and Control System parts or components, and trouble shooting, that terminates when the Robot and Control System has been reset to its original project functional condition.

Putting Out of Service and Dismantling

Putting out of service defines the activities involved in the mechanical and electrical removal of the Robot and Control System from a production unit or from an environment in which it was under study.

Dismantling consists of the demolition and dismantling of the components that make up the Robot and Control System.

Integrator

The integrator is the professional expert responsible for the installation and putting into service of the Robot and Control System.

Incorrect Use

Incorrect use is when the system is used in a manner other than that specified in the Technical Documentation.

Range of Action

The robot range of action is the enveloping volume of the area occupied by the robot and its fixtures during movement in space.

1.2.3 Applicability

These Specifications are to be applied when executing the following activities:

- Installation and Putting into Service
- Programming Mode
- Auto / Remote Automatic Mode
- Robot axes release
- Maintenance and Repairs
- Putting Out of Service and Dismantling

1.2.4 Operating Modes

Installation and Putting into Service

- Putting into service is only possible when the Robot and Control System has been correctly and completely installed.
- The system installation and putting into service is exclusively the task of the authorised personnel.
- The system installation and putting into service is only permitted inside a protected area of an adequate size to house the robot and the fixtures it is outfitted with, without passing beyond the safety barriers. It is also necessary to check that under normal robot movement conditions there is no collision with parts inside the protected area (structural columns, power supply lines, etc.) or with the barriers. If necessary, limit the robot working areas with mechanical hard stop (see optional assemblies).
- Any fixed robot control protections are to be located outside the protected area and in a point where there is a full view of the robot movements.
- The robot installation area is to be as free as possible from materials that could impede or limit visibility.
- During installation the robot and the Control Unit are to be handled as described in the product Technical Documentation; if lifting is necessary, check that the eye-bolts are fixed securely and use only adequate slings and equipment.
- Secure the robot to the support, with all the bolts and pins foreseen, tightened to the torque indicated in the product Technical Documentation.
- If present, remove the fastening brackets from the axes and check that the fixing of the robot fixture is secured correctly.
- Check that the robot guards are correctly secured and that there are no moving or loose parts. Check that the Control Unit components are intact.
- If applicable, connect the robot pneumatic system to the air distribution line paying attention to set the system to the specified pressure value: a wrong setting of the pressure system influences correct robot movement.
- Install filters on the pneumatic system to collect any condensation.
- Install the Control Unit outside the protected area: the Control Unit is not to be used to form part of the fencing.
- Check that the voltage value of the mains is consistent with that indicated on the plate of the Control Unit.
- Before electrically connecting the Control Unit, check that the circuit breaker on the mains is locked in open position.
- Connection between the Control Unit and the three-phase supply mains at the works, is to be with a four-pole (3 phases + earth) armoured cable dimensioned appropriately for the power installed on the Control Unit. See the product Technical Documentation.
- The power supply cable is to enter the Control Unit through the specific fairlead and be properly clamped.
- Connect the earth conductor (PE) then connect the power conductors to the main switch.

- Connect the power supply cable, first connecting the earth conductor to the circuit breaker on the mains line, after checking with a tester that the circuit breaker terminals are not powered. Connect the cable armouring to the earth.
- Connect the signals and power cables between the Control Unit and the robot.
- Connect the robot to earth or to the Control Unit or to a nearby earth socket.
- Check that the Control Unit door (or doors) is/are locked with the key.
- A wrong connection of the connectors could cause permanent damage to the Control Unit components.
- The C5G Control Unit manages internally the main safety interlocks (gates, enabling pushbuttons, etc.). Connect the C5G Control Unit safety interlocks to the line safety circuits, taking care to connect them as required by the Safety standards. The safety of the interlock signals coming from the transfer line (emergency stop, gates safety devices etc) i.e. the realisation of correct and safe circuits, is the responsibility of the Robot and Control System integrator.



In the cell/line emergency stop circuit the contacts must be included of the control unit emergency stop buttons, which are on X30. The push buttons are not interlocked in the emergency stop circuit of the Control Unit.

- The safety of the system cannot be guaranteed if these interlocks are wrongly executed, incomplete or missing.
- The safety circuit executes a controlled stop (IEC 60204-1 , class 1 stop) for the safety inputs Auto Stop/ General Stop and Emergency Stop. The controlled stop is only active in Automatic states; in Programming the power is cut out (power contactors open) immediately. The procedure for the selection of the controlled stop time (that can be set on SDM board) is contained in the Installation manual .
- When preparing protection barriers, especially light barriers and access doors, bear in mind that the robot stop times and distances are according to the stop category (0 or 1) and the weight of the robot.



Check that the controlled stop time is consistent with the type of Robot connected to the Control Unit. The stop time is selected using selector switches SW1 and SW2 on the SDM board.

- Check that the environment and working conditions are within the range specified in the specific product Technical Documentation.
- The calibration operations are to be carried out with great care, as indicated in the Technical Documentation of the specific product, and are to be concluded checking the correct position of the machine.
- To load or update the system software (for example after replacing boards), use only the original software handed over by COMAU Robotics & Service. Scrupulously follow the system software uploading procedure described in the Technical Documentation supplied with the specific product. After uploading, always make some tests moving the robot at slow speed and remaining outside the protected area.
- Check that the barriers of the protected area are correctly positioned.

Programming Mode

- The robot is only to be programmed by the authorised personnel.
- Before starting to program, the operator must check the **Robot System (Robot and Control System)** to make sure that there are no potentially hazardous irregular conditions, and that there is nobody inside the protected area.
- When possible the programming should be controlled from outside the protected area.
- Before operating inside the **Protected Area**, the operator must make sure from outside that all the necessary protections and safety devices are present and in working order, and especially that the hand-held programming unit functions correctly (slow speed, emergency stop, enabling device, etc.).
- During the programming session, only the operator with the hand-held terminal is allowed inside the **Protected Area**.
- If the presence of a second operator in the working area is necessary when checking the program, this person must have an enabling device interlocked with the safety devices.
- Activation of the motors (Drive On) is always to be controlled from a position outside the range of the robot, after checking that there is nobody in the area involved. The Drive On operation is concluded when the relevant machine status indication is shown.
- When programming, the operator is to keep at a distance from the robot to be able to avoid any irregular machine movements, and in any case in a position to avoid the risk of being trapped between the robot and structural parts (columns, barriers, etc.), or between movable parts of the actual robot.
- When programming, the operator is to avoid remaining in a position where parts of the robot, pulled by gravity, could execute downward movements, or move upwards or sideways (when installed on a sloped plane).
- Testing a programmed cycle at working speed with the operator inside the protected area, in some situations where a close visual check is necessary, is only to be carried out after a complete test cycle at slow speed has been executed. The test is to be controlled from a safe distance.
- Special attention is to be paid when programming using the hand-held terminal: in this situation, although all the hardware and software safety devices are active, the robot movement depends on the operator.
- During the first running of a new program, the robot may move along a path that is not the one expected.
- The modification of program steps (such as moving by a step from one point to another of the flow, wrong recording of a step, modification of the robot position out of the path that links two steps of the program), could give rise to movements not envisaged by the operator when testing the program.
- In both cases operate cautiously, always remaining out of the robot's range of action and test the cycle at slow speed.

Auto / Remote Automatic Mode

- The activation of the automatic operation (AUTO and REMOTE states) is only to be executed with the [Robot System \(Robot and Control System\)](#) integrated inside an area with safety barriers properly interlocked, as specified by Safety Standards currently in force in the Country where the installation takes place.
- Before starting the automatic mode the operator is to check the Robot and Control System and the protected area to make sure there are no potentially hazardous irregular conditions.
- The operator can only activate automatic operation after having checked:
 - that the Robot and Control System is not in maintenance or being repaired;
 - the safety barriers are correctly positioned;
 - that there is nobody inside the protected area;
 - that the Control Unit doors are closed and locked;
 - that the safety devices (emergency stop, safety barrier devices) are functioning;
- Special attention is to be paid when selecting the automatic-remote mode, where the line PLC can perform automatic operations to switch on motors and start the program.

Robot axes release

- In the absence of motive power, the robot axes movement is possible by means of optional release devices and suitable lifting devices. Such devices only enable the brake deactivation of each axis. In this case, all the system safety devices (including the emergency stop and the enable button) are cut out; also the robot axes can move upwards or downwards because of the force generated by the balancing system, or the force of gravity.



Before using the manual release devices, it is strongly recommended to sling the robot, or hook to an overhead travelling crane.

- Enabling the brake releasing device may cause the axes falling due to gravity as well as possible impacts due to an incorrect restoration, after applying the brake releasing module. The procedure for the correct usage of the brake releasing device (both for the integrated one and module one) is to be found in the maintenance manuals.
- When the motion is enabled again following the interruption of an unfinished MOVE, the track recovery typical function may generate unpredictable paths that may imply the risk of impact. This same condition arises at the next automatic cycle restarting. Avoid moving the Robot to positions that are far away from the ones provided for the motion restart; alternatively disable the outstanding MOVE programmes and/or instructions.

Maintenance and Repairs

- When assembled in COMAU Robotics & Service, the robot is supplied with lubricant that does not contain substances harmful to health, however, in some cases, repeated and prolonged exposure to the product could cause skin irritation, or if swallowed, indisposition.

First Aid. Contact with the eyes or the skin: wash the contaminated zones with abundant water; if the irritation persists, consult a doctor.

If swallowed, do not provoke vomiting or take anything by mouth, see a doctor as soon as possible.

- Maintenance, trouble-shooting and repairs are only to be carried out by authorised personnel.
- When carrying out maintenance and repairs, the specific warning sign is to be placed on the control panel of the Control Unit, stating that maintenance is in progress and it is only to be removed after the operation has been completely finished - even if it should be temporarily suspended.
- Maintenance operations and replacement of components or the Control Unit are to be carried out with the main switch in open position and locked with a padlock.
- Even if the Control Unit is not powered (main switch open), there may be interconnected voltages coming from connections to peripheral units or external power sources (e.g. 24 Vdc inputs/outputs). Cut out external sources when operating on parts of the system that are involved.
- Removal of panels, protection shields, grids, etc. is only allowed with the main switch open and padlocked.
- Faulty components are to be replaced with others having the same code, or equivalent components defined by COMAU Robotics & Service.



After replacement of the SDM module, check on the new module that the setting of the stop time on selector switches SW1 and SW2 is consistent with the type of Robot connected to the Control Unit.

- Trouble-shooting and maintenance activities are to be executed, when possible, outside the protected area.
- Trouble-shooting executed on the control is to be carried out, when possible without power supply.
- Should it be necessary, during trouble-shooting, to intervene with the Control Unit powered, all the precautions specified by Safety Standards are to be observed when operating with hazardous voltages present.
- Trouble-shooting on the robot is to be carried out with the power supply cut out (Drive off).
- At the end of the maintenance and trouble-shooting operations, all deactivated safety devices are to be reset (panels, protection shields, interlocks, etc.).
- Maintenance, repairs and trouble-shooting operations are to be concluded checking the correct operation of the **Robot System (Robot and Control System)** and all the safety devices, executed from outside the protected area.
- When loading the software (for example after replacing electronic boards) the original software handed over by COMAU Robotics & Service is to be used. Scrupulously follow the system software loading procedure described in the specific product Technical Documentation; after loading always run a test cycle to make sure, remaining outside the protected area
- Disassembly of robot components (motors, balancing cylinders, etc.) may cause uncontrolled movements of the axes in any direction: before starting a disassembly procedure, consult the warning plates applied to the robot and the Technical Documentation supplied.
- It is strictly forbidden to remove the protective covering of the robot springs.

Putting Out of Service and Dismantling

- Putting out of service and dismantling the Robot and Control System is only to be carried out by [Authorised Personnel](#).
- Bring the robot to transport position and fit the axis clamping brackets (where applicable) consulting the plate applied on the robot and the robot Technical Documentation.
- Before stating to put out of service, the mains voltage to the Control Unit must be cut out (switch off the circuit breaker on the mains distribution line and lock it in open position).
- After using the specific instrument to check there is no voltage on the terminals, disconnect the power supply cable from the circuit breaker on the distribution line, first disconnecting the power conductors, then the earth. Disconnect the power supply cable from the Control Unit and remove it.
- First disconnect the connection cables between the robot and the Control Unit, then the earth cable.
- If present, disconnect the robot pneumatic system from the air distribution line.
- Check that the robot is properly balanced and if necessary sling it correctly, then remove the robot securing bolts from the support.
- Remove the robot and the Control Unit from the work area, applying the rules indicated in the products Technical Documentation; if lifting is necessary, check the correct fastening of the eye-bolts and use appropriate slings and equipment only.
- Before starting dismantling operations (disassembly, demolition and disposal) of the Robot and Control System components, contact COMAU Robotics & Service, or one of its branches, who will indicate, according to the type of robot and Control Unit, the operating methods in accordance with safety principles and safeguarding the environment.
- The waste disposal operations are to be carried out complying with the legislation of the country where the Robot and Control System is installed.

1.2.5 Performance

The performances below shall be considered before installing the robot system:

- Stop distances
- Mission time (typ. case).

Stop distances

- With Robot in programming modality (T1), if you press the stop pushbutton (red mushroom-shaped one on WiTP) in category 0 (according to the standard EN60204-1), you will obtain:

Tab. 1.1 - Stop spaces in programming modality (T1)

Mode	Expected speed	Case	Stopping time	Stopping space
T1	250 mm/s	Nominal	120 ms	30 mm
		Limit	500 ms	125 mm

Tab. 1.2 - Safety electronics reaction time in programming modality (T1)

Mode	Expected speed	Case	Stopping time
T1	250 mm/s	For the safety inputs of the SDM module (e.g. stop pushbutton of TP in wired version)	150 ms
		For the stop stop and enabling device inputs from the TP in wireless version, when the safety wire transmission is active.	
		For the time-out of stop input and enabling device from TP in wireless version, when the safety wire transmission is lost or interrupted.	350 ms

- Considering the Robot in automatic modality, under full extension, full load and maximum speed conditions, if you press the stop pushbutton (red mushroom-shaped one on WiTP) in category 1 (according to norm EN60204-1) you will trigger the Robot complete stop with controlled deceleration ramp.
Example: for Robot NJ 370-2.7 you will obtain the complete stop in about 85 ° motion, that correspond to about 3000 mm movement measured on TCP flange. Under the said conditions, the stopping time for Robot NJ 370-2.7 is equal to 1,5 seconds.
- For each Robot type the limit stop time can be required to COMAU Robotics g

Mission time (typ. case)

- We remind you that the safety system efficiency covering is equal to 20 years (**mission time** of safety-related parts of control systems (SRP/CS), according to EN ISO 13849-1).

2. INTRODUCTION TO PDL2

- [Peculiarities of PDL2 Language](#)
- [Syntax notation](#)
- [Language components](#)
- [Statements](#)
- [Program structure](#)
- [Units of measure](#)

2.1 Peculiarities of PDL2 Language

PDL2 is a Pascal-like programming language, used to write user programs and applications, consisting of statements for:

- moving robot arms;
- following the program flow (if, while, repeat, for,...);
- sending and receiving information on files (read, write open file), devices, serial protocols;
- monitoring up to 255 events and states per program, including input/output port ones;
- implementing error handling.

In addition, it includes almost 200 predefined routines and functions with several purposes useful to diversify the handling of arm movement and process control.

It allows to simultaneously execute more than 250 multiple programs to handle all process control aspects of an application.

A single Controller Unit can also control multiple robotic arms and related equipment.

The language is open to handle up to 4 arms, with 10 axes (4 of them can be auxiliary), each one depending from the hardware configuration.

An integrated editing environment, including syntax checking and statements execution, is available on the robot Controller.

PDL2 programs are divided into two categories, depending on the holdable/non-holdable attribute (see also [par. 10.35 PROGRAM Statement on page 208](#)):

- holdable programs (indicated by the HOLD attribute) are controlled by START and HOLD buttons. Generally, holdable programs include motion, but that is not a requirement;
- non-holdable programs (indicated by the NOHOLD attribute) are not controlled by START and HOLD buttons. Generally, they are used as process control programs. Non-holdable programs cannot contain motion statements, however, they can use positional variables for other purposes. The motion statements which are not allowed in non-holdable programs are RESUME, CANCEL, LOCK, UNLOCK, and MOVE.

2.2 Syntax notation

This manual uses the following notation to represent the syntax of PDL2 statements: optional items are enclosed in angle brackets. For example, the description:

- item1 <item2> item3

has the following possible results:

```
item1 item3
item1 item2 item3
```

- items that occur one or more times are followed by three dots. For example, the description:

```
item1 item2...
```

has the following possible results:

```
item1 item2
item1 item2 item2
item1 item2 item2 item2 etc.
```

- vertical bars separate choices. If at least one item must be chosen, the whole set of choices is enclosed in double vertical bars. For example, the description:

```
|| item1 | item2 ||
```

has the following possible results:

```
item1
item2
```

Combinations of these notations provide powerful, but sometimes tricky, syntax descriptions. They are extremely helpful in understanding what can be done in the PDL2 language. A few examples follow

Description	Possible Results
item1 <item2 item3>	item1 item1 item2 item1 item3
item1 <item2 item3>...	item1 item1 item2 item1 item3 item1 item3 item2 etc.
item1 item2 item3 ...	item1 item2 item3 item2 item2 item1 item3 item1 item3 item2 etc.

Note that when the repeating dots come right after the optional brackets, the items inside the brackets can be repeated zero or more times. However, when they come after double vertical bars, the items inside the bars can be repeated one or more times.

2.3 Language components

This section explains the components of the PDL2 language:

- [Character set](#)
- [Reserved words, Symbols, and Operators](#)
- [Predefined Identifiers](#)
- [User-defined Identifiers](#)
- [Blank space](#)
- [Comments](#)

2.3.1 Character set

PDL2 recognizes the characters shown in [Tab. 2.1](#).

Tab. 2.1 - PDL2 Character Set

Letters:	a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Symbols:	@ < > = / * + - _ , ; . # \$ [] % { } \ : ! ()
Special Characters:	blank (space), tab

PDL2 does not distinguish between uppercase and lowercase letters.

2.3.2 Reserved words, Symbols, and Operators

A reserved word, symbol, or operator is one that has a special and unchangeable meaning in PDL2. Words identify sections of a program, data types, and key words in a statement. Symbols usually punctuate a statement. Operators indicate a calculation or comparison.

[Tab. 2.2](#) lists the reserved words, symbols, and operators.

Tab. 2.2 - Reserved Words, Symbols, and Operators

ABORT	ABOUT	ACTIVATE	ACTIVATES
ADVANCE	AFTER	ALARM	ALL
ALONG	AND	ANY	ANYERROR
ARM	ARRAY	ASSERT	AT
ATTACH	AWAY	BEFORE	BEGIN
BOOLEAN	BREAK	BY	BYPASS
CALL	CALLS	CANCEL	CASE
CATCH	CLASS	CLOSE	CONDITION
CONNECT	CONST	CONTINUE	CURRENT

Tab. 2.2 - Reserved Words, Symbols, and Operators (Continued)

CYCLE	CYCLES	DEACTIVATE	DEACTIVATES
DECODE	DELAY	DETACH	DISABLE
DISTANCE	DIV	DO	DOWNTO
DV_CNTRL	ELSE	ENABLE	ENCODE
END	ENDCONDITION	ENDFOR	ENDIF
ENDMOVE	ENDNODEDEF	ENDOPEN	ENDRECORD
ENDSELECT	ENDTRY	ENDWHILE	ERRORCLASS
ERRONUM	EVENT	EXECS	EXIT
EXPORTED	FILE	FINAL	FOR
FROM	GOTO	GOTOS	HAND
HOLD	IF	IMPORT	IN
INTEGER	INTERRUPT	JOINTPOS	LOCK
LONGJUMP	MC	MJ	ML
MOD	MOVE	MOVEFLY	MV
NEAR	NL	NODATA	NODEDEF
NODISABLE	NOHOLD	NOSAVE	NOT
NOTEACH	OF	OPEN	OR
PATH	PAUSE	PAUSES	PERCENT
PLC	POSITION	POWERUP	PRIORITY
PROGRAM	PROG_ARM	PULSE	PURGE
RAISE	READ	REAL	RECORD
RELATIVE	RELAX	REPEAT	RESUME
RETRY	RETURN	ROL	ROR
ROUTINE	SCAN	SEGMENT	SELECT
SEMAPHORE	SETJUMP	SHL	SHR
SIGNAL	SKIP	STACK	START
STEP	STOP	STRING	SYNCMOVE
SYNCMOVEFLY	THEN	TIL	TIME
TO	TRY	TYPE	UAL
UNLOCK	UNPAUSE	UNPAUSES	UNTIL
VAR	VECTOR	VIA	VOID
WAIT	WHEN	WHILE	WINDOW
WITH	WRITE	XOR	XTNDPOS
YELD	[]	{
}	()	<
>	<=	>=	<>
=	+	-	*
/	**	,	.
..	:	::	:=
;	#	@	+=
- =			

2.3.3 Predefined Identifiers

Predefined identifiers make up the remainder of words that are part of the PDL2 language. They name constants, variables, fields, and routines that PDL2 already understands. Predefined identifiers differ from reserved words in that the programmer can redefine the meaning of a predefined identifier that is not used in the program. Predefined constants are identifiers that have preassigned values associated with them. A predefined constant identifier can be used in a program instead of the value. The following [Tab. 2.3 - Predefined Constants](#) lists the available predefined constants.

Tab. 2.3 - Predefined Constants

ADV	BASE	CIRCULAR	COARSE
COLL_DSBL	COLL_HIGH	COLL_LOW	COLL_MANUAL
COLL_MEDIUM	COLL_USER1..10	COM_ASCII	COM_BD110
COM_BD300	COM_BD1200	COM_BD2400	COM_BD4800
COM_BD9600	COM_BD19200	COM_BD38400	COM_BD57600
COM_BD115200	COM_BIT7	COM_BIT8	COM_CHAR
COM_CHARNO	COM_PAR_EVEN	COM_PAR_NO	COM_PAR_ODD
COM_PASAL	COM_RDAHD	COM_RDAHD_NO	COM_STOP1
COM_STOP1_5	COM_STOP2	COM_XSYNC	COM_XSYNC_NO
CONV_INIT	CONV_INIT_ON	CONV_OFF	CONV_ON
DT_ABOOLEAN	DT_AINTEGER	DT_AJOINTPOS	DT_ANODE
DT_APOSITION	DT_AREAL	DT_ARCORD	DT_ASEMAPHORE
DT_ASTRING	DT_AVECTOR	DT_AXTNPOS	DT_BOOLEAN
DT_INTEGER	DT_JOINTPOS	DT_NODE	DT_PATH
DT_POSITION	DT_REAL	DT_RECORD	DT_SEMAPHORE
DT_STRING	DT_VECTOR	DT_XTNPOS	EC_BYPASS
EC_COND	EC_DISP	EC_ELOG	EC_FILE
EC_MATH	EC_PIO	EC_PLA	EC_PROG
EC_RLL	EC_SYS	EC_SYS_C	EC_TRAP
EC_USR1	EC_USR2	ERR_ABORT	ERR_ACK
ERR_CANCEL	ERR_FALSE	ERR_IGNORE	ERR_NO
ERR_OFF	ERR_OK	ERR_ON	ERR_RESET
ERR_RETRY	ERR_SKIP	ERR_TRUE	ERR_YES
EUL_WORLD	FALSE	FINE	FLY
FLY_AUTO	FLY_CART	FLY_FROM	FLY_NORM
FLY_PASS	FLY_TOL	IR_CYLINDER	IR_FORBIDDEN
IR_JOINT	IR_MONITORED	IR_NOT_PRESENT	IR_PARALLELEPIPED
IR_PLANE	IR_PRESENCE	IR_SPHERE	JNT_COARSE
JNT_FINE	JOINT	JPAD_LIN	JPAD_ROT
LANG_CS	LANG_DE	LANG_EN	LANG_FR
LANG_IT	LANG_PL	LANG_PO	LANG_RU
LANG_SP	LANG_TR	LANG_ZH	LINEAR
LUN_CRT	LUN_NULL	LUN_SIO	LUN_TP
MAXINT	MININT	NOADV	NOFLY

Tab. 2.3 - Predefined Constants

NOSETTLE	OFF	ON	ON_MV
PDV_CRT	PDV_TP	RPY_WORLD	RS_TRAJ
RS_WORLD	SCRN_ALARM	SCRN_APPL	SCRN_CLR_CHR
SCRN_CLR_DEL	SCRN_CLR_Rem	SCRN_DATA	SCRN_EDIT
SCRN_FILE	SCRN_IDE	SCRN_IO	SCRN_LOGIN
SCRN_MOTION	SCRN_PROG	SCRN_SERVICE	SCRN_SETUP
SCRN_SYS	SCRN_TPINT	SCRN_USER	SCRN_VP2
SEG_VIA	SPD_AUX1	SPD_AUX2	SPD_AUX3
SPD_AUX4	SPD_AZI	SPD_CONST	SPD_ELV
SPD_FIRST	SPD_JNT	SPD_LIN	SPD_PGOV
SPD_ROLL	SPD_ROT	SPD_SECOND	SPD_SM4C
SPD_SPN	SPD_THIRD	STR_COLL	STR_COMP
STR_COMPOSE	STR_DECOMPOSE	STR_LWR	STR_OPER_ADLER
STR_OPER_BXOR	STR_OPER_CRC32	STR_TRIM	STR_UPR
TOOL	TRUE	UFRAME	WIN_BLACK
WIN_BLINK_OFF	WIN_BLINK_ON	WIN_BLUE	WIN_BOLD_OFF
WIN_BOLD_ON	WIN_CLR_ALL	WIN_CLR_BOLN	WIN_CLR_BOW
WIN_CLR_EOLN	WIN_CLR_EOW	WIN_CLR_LINE	WIN_CRSR_OFF
WIN_CRSR_ON	WIN_CYAN	WIN_DARKGREY	WIN_GREEN
WIN_GREY	WIN_LIGHTGREY	WIN_MAGENTA	WIN_ORANGE
WIN_PINK	WIN_RED	WIN_REVERSE	WIN_SCROLL
WIN_WHITE	WIN_WRAP	WIN_YELLOW	WRIST_JNT
ZERO			

Predefined variables have preassigned data types and uses All predefined variables begin with a dollar sign (\$). [Predefined Variables List](#) chapter is an alphabetical reference of predefined variables.

Predefined fields provide access to individual components of structured data types. Each field is explained in [Data Representation](#) chapter under the data type to which it relates.

Predefined routines, also called built-in routines, are provided to handle commonly required programming tasks. [BUILT-IN Routines List](#) chapter is an alphabetical reference of built-in routines.

2.3.4 User-defined Identifiers

User-defined identifiers are the names a programmer chooses to identify the following:

- programs;
- variables;
- constants;
- routines;
- labels;
- types;

- fields.

A user-defined identifier must start with a letter. It can contain any number of letters, digits, and underscore (_) characters. A user-defined identifier can have only one meaning within a given scope. The scope of an identifier indicates the section of a program that can reference the identifier.

Program identifiers are contained in a separate scope which means the user can define a variable having the same name as a program.

A user-defined variable is a name that can represent any value of a particular type. The programmer declares a variable by associating a name with a data type. That name can then be used in the program to represent any value of that type.

A user-defined constant is a name that represents a specific value. The programmer declares a constant by equating a name to a value. That name can then be used in the program to represent the value. [Data Representation](#) chapter explains variable and constant declarations.

A user-defined routine is a set of instructions represented by a single name. The programmer can define routines that handle specific parts of the overall job. The routine name can be used in a program to represent the actual instructions. [Routines](#) chapter describes user-defined routines.

User-defined labels are used to mark the destination of a GOTO statement. [Execution Control](#) chapter describes the use of labels and GOTO statements.

A user-defined type is a set of field definitions represented by a single name. The programmer declares a type in order to define a new data type which is a sequence of existing data types. The type name can then be used in the declaration of a variable or routine. [Data Representation](#) chapter explains type and field declarations.

2.4 Statements

PDL2 programs are composed of statements. Statements are a combination of the following:

- reserved words, symbols, and operators;
- predefined identifiers;
- user-defined identifiers.

Statements must be syntactically correct; that is, they must be constructed according to the syntax rules of PDL2. The program editor helps provide the correct syntax as statements are entered. [Statements List](#) chapter contains an alphabetical reference of PDL2 statements.

2.4.1 Blank space

Blank spaces separate reserved words and identifiers. Blanks can be used, but are not required, between operators and their operands (a + b or a+b). Blanks can also be used to indent statements. However, the editor automatically produces standard spacing and indentation, regardless of what the programmer uses.

2.4.2 Comments

A comment is text in a program that is not part of the program instructions. Comments have no effect on how the controller executes statements. A comment begins with two hyphens (—). The controller will ignore any text that follows the two hyphens, up to a maximum length of 255 characters. Typically, comments are used by a programmer to

explain something about the program.

2.5 Program structure

The structure of a program is as follows:

```
PROGRAM name <attributes>
    <import statements>
    <constant, variable, and type declarations>
    <routine declarations>
BEGIN <CYCLE>
    <executable statements>
END name
```

A program starts with the PROGRAM statement. This statement identifies the program with a user-defined *name*. The same name identifies the file in which the program is stored. Optional program attributes, explained with the PROGRAM statement in [Statements List](#) chapter, can also be included.

Programs are divided into a declaration section and an executable section. The declaration section is a list of all the user-defined data items and routines the program will use. The executable section is a list of statements the controller will execute to perform a task.

The BEGIN statement separates the declaration section from the executable section. The programmer can include the CYCLE option with the BEGIN statement to create a continuous cycle. The END statement marks the end of the program and also includes the program name.

In this manual, reserved words and predefined identifiers are capitalized, user-defined identifiers are italicized, and optional items are enclosed in angle brackets <>. A complete description of syntax notation is provided at the end of this chapter.

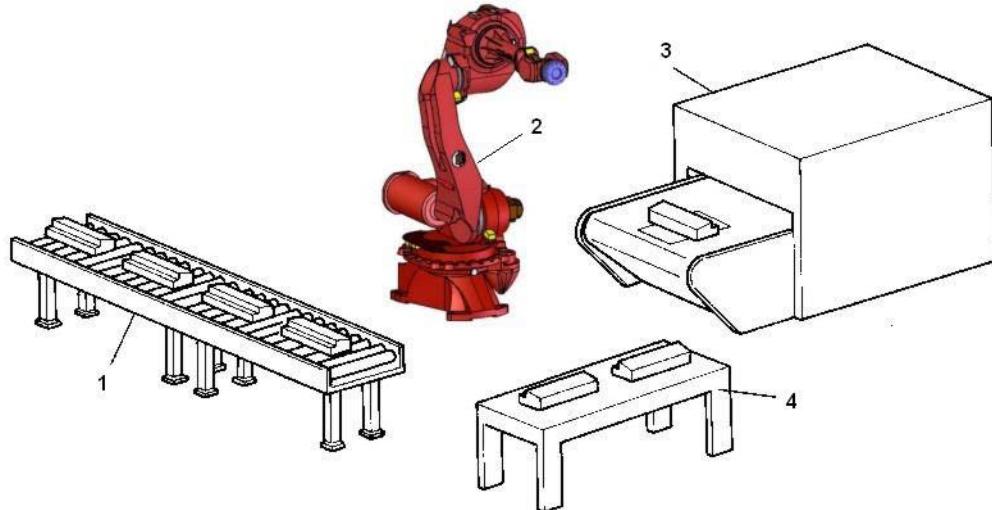
2.6 Program example

The following PDL2 program transfers parts from a feeder to a work table or a discard bin, as shown in [Fig. 2.1 - Pack Program Work Area](#). Digital input signals indicate when the feeder is ready and whether or not the part should be discarded.

```
PROGRAM pack
VAR
    perch, feeder, table, discard : POSITION
BEGIN CYCLE
    MOVE TO perch
    OPEN HAND 1
    WAIT FOR $DIN[1] = ON
    -- signals feeder ready
    MOVE TO feeder
    CLOSE HAND 1
    IF $DIN[2] = OFF THEN
    -- determines if good part
        MOVE TO table
    ELSE
        MOVE TO discard
```

```

ENDIF
OPEN HAND 1
-- drop part on table or in bin
END pack
  
```

Fig. 2.1 - Pack Program Work Area

1. Feeder
2. Robot
3. Discard Bin
4. Table

2.7 Units of measure

PDL2 uses the units of measure shown in [Tab. 2.4 - PDL2 Units](#).

Tab. 2.4 - PDL2 Units

Distance:	millimeters(mm)
Time:	milliseconds(ms)
Angles:	degrees (°)
Linear Velocity:	meters/second (m/s)
Angular Velocity:	radians/second (rad/s)
Current:	ampere (A)
Encoder/Resolver Data:	revolutions (2 ms)

3. DATA REPRESENTATION

This chapter explains each PDL2 data type, how data is declared, and how it can be manipulated within a program.

[Data Types](#) determine the following:

- the kinds of values associated with a data item;
- the operations that can be performed on the data.

PDL2 programs can include the following kinds of data items:

- VARIABLES, representing values that can change;
- CONSTANTS, representing values that cannot change;
- LITERALS, actual values.

The following information are supplied:

- [Data Types](#)
- [Declarations](#)
- [Expressions](#)
- [Assignment Statement](#)
- [Typecasting](#)

Variables and constants are defined by an identifier and a data type.

Declaring a variable associates an identifier with a data type.

Different values of that type can be assigned to the identifier throughout the program (unless they have been declared with the CONST attribute - see [par. – CONST attribute on page 66](#)). VARIABLES can be declared to be of any data type.

Declaring a CONSTANT associates a value with an identifier. That value cannot be changed within the program. The data type of the identifier is determined by the value. INTEGER, REAL, BOOLEAN, and STRING values can be associated with constant identifiers.

LITERAL values are actual values used in the program. They can be INTEGER, REAL, or STRING values.

PDL2 uses expressions to manipulate data in a program. Expressions are composed of operands and operators. Operands are the data items being manipulated. Operators indicate what kind of manipulation is performed.

3.1 Data Types

This section describes the different data types that are available in PDL2 and lists the operations that can be performed on each data type.

A detailed description follows about:

- [INTEGER](#)
- [REAL](#)
- [BOOLEAN](#)
- [STRING](#)

- **ARRAY**
- **RECORD**
- **VECTOR**
- **POSITION**
- **JOINTPOS**
- **XTNDPOS**
- **NODE**
- **PATH**
- **SEMAPHORE**

3.1.1 INTEGER

The INTEGER data type represents whole number values in the range -2147483647 through +2147483647. The following predefined constants represent the maximum and minimum INTEGER values:

- **MAXINT;**
- **MININT.**

An INTEGER can be represented as decimal (base 10), octal (base 8), hexadecimal (base 16) or binary (base 2). The default base for INTEGERs is decimal. To represent a based INTEGER literal, precede the number with 0o to specify octal (0o72), Ox to specify hexadecimal (0xFF), or Ob to specify binary (0b11011).

PDL2 can perform the following operations on INTEGER data:

- arithmetic (+, -, *, /, DIV, MOD, **, +=, -=);
- relational (<, >, =, <>, <=, >=);
- bitwise (AND, OR, XOR, NOT, SHR, SHL, ROR, ROL).

The += and -= operators are used for incrementing and decrementing integer program variables. They are not permitted to be used on system variables.

The amount of increment can be expressed by a constant or by another integer variable. For example:

```
VAR i, j: INTEGER
i += 5 -- It is equivalent to i:=i+5
i -= j -- It is equivalent to i:=i-j
```

This operator can also be used in condition actions.

At run time, an INTEGER PDL2 variable will assume the value of uninitialized if it becomes greater than MAXINT.

In addition, PDL2 provides built-in routines to access individual bits of an INTEGER value and to perform other common INTEGER functions (refer to [BUILT-IN Routines List](#) chapter).

3.1.2 REAL

The REAL data type represents numeric values that include a decimal point and a fractional part or numbers expressed in scientific notation.

[Fig. 3.1 - Range of REAL Data](#) shows the range for REAL data.

Fig. 3.1 - Range of REAL Data



PDL2 can perform the following operations on REAL data:

- arithmetic (+, -, *, /, **);
- relational (<, >, =, <>, <=, >=).

In addition, PDL2 provides built-in routines to perform other common REAL functions (refer to the [BUILT-IN Routines List](#) chapter).

REAL values used in a PDL2 program are always displayed using eight significant digits.

3.1.3 BOOLEAN

The BOOLEAN data type represents the Boolean predefined constants TRUE (ON) and FALSE (OFF).

PDL2 can perform the following operations on BOOLEAN data:

- relational (=, <>);
- Boolean (AND, OR, XOR, NOT).

3.1.4 STRING

The STRING data type represents a series of characters (either ASCII or UNICODE), treated as a single unit of data.

Single quotes mark the beginning and the end of an ASCII string value.

For example:

```
'this is an ASCII string value'
```

Double quotes mark the beginning and the end of a UNICODE string value.

For example:

```
"this is a UNICODE string value"
```

```
"程序机构可以进行编辑并示教开发程序 "
```

As far as ASCII strings, in addition to printable characters, they can also contain control sequences.

An ASCII control sequence consists of a backslash (\) followed by any three digit ASCII code.

For example:

```
ASCII Literal: 'ASCII code 126 is a tilde: \126'
ASCII Value:      ASCII code 126 is a tilde: ~
```

As far as UNICODE, since it includes a unique code for any type of existing symbols, strings can contain any type of worldwide used characters.

Each of them can also be indicated by means of \u followed by its UNICODE code.

For example:

```
UNICODE Literal: "UNICODE 8A89 is \u0x8A89"
```

```
UNICODE value:      UNICODE 8A89 is 赞
```

All characters in a STRING value, not represented in the above formats, are replaced with blanks.

To produce either the backslash (\) or the single ('') or double ("") quotes characters in a STRING literal, use two in a row.

For example:

Literal: 'Single quote'
 Value: Single quote '

Literal: 'Back slash \\'
 Value: Back slash \

Literal: "Double quote""
 Value: Double quote "

The actual length of a string value can be from 0 to 2048 bytes.

An actual length of 0 represents an empty STRING value.

PDL2 can perform the following operations on STRING data:

- relational (<, >, =, <>, <=, >=)

In addition, PDL2 provides built-in routines to perform common STRING manipulations (refer to [par. String Built-In Routines on page 229](#)).



Note that for UNICODE strings, the required length is double + 2, compared with ASCII strings.

Each UNICODE character is internally represented by means of 2 bytes, whereas each ASCII character is represented by means of 1 byte only.

3.1.5 ARRAY

The ARRAY data type represents an ordered collection of data items, all of the same type.

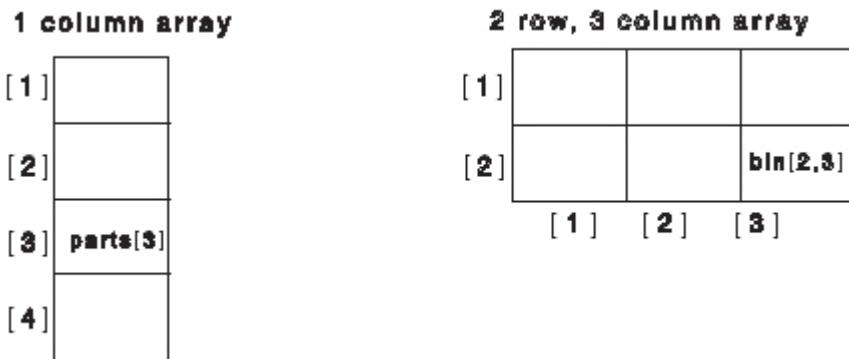
The programmer can declare that type to be one of the following:

INTEGER	VECTOR
REAL	POSITION
BOOLEAN	JOINTPOS
STRING	XTNDPOS
RECORD	SEMAPHORE

The programmer can declare an ARRAY to have one or two dimensions.

The programmer also declares the maximum number of items for each dimension, up to 65535 for each. (Note: The actual size might be limited by the amount of available system memory.)

Fig. 3.2 - Representing Arrays



Individual items in an ARRAY are referenced by index numbers.

For one dimensional arrays, a single INTEGER expression represents the row position of the item in the ARRAY. For two dimensional arrays, two INTEGER expressions, separated by a comma, represent the row and column position. Index numbers follow the ARRAY name and are enclosed in square brackets.

In Fig. 3.2 - Representing Arrays, the third item in the one dimensional ARRAY is referenced as *parts[3]*. The item in the second row, third column in the two dimensional array is referenced as *bin[2,3]*.

All of the operations that are available for a particular data type can be performed on individual ARRAY items of that type. An entire ARRAY can be used as an argument in a routine call or in an assignment statement. When using arrays in assignment statements, both arrays must be of the same data type, size, and dimension. SEMAPHORE arrays cannot be used in an assignment statement.

3.1.6 RECORD

The RECORD data type represents a collection of one or more data items grouped together using a single name. Each item of a record is called a field and can be of any PDL2 data type except SEMAPHORE, RECORD, NODE, or PATH.

The predefined data types VECTOR, POSITION, and XTNDPOS are examples of record types. The user can define new record data types in the TYPE section of a program.

A RECORD type definition creates a user-defined data type that is available to the entire system. This means that it is possible to have conflicts between RECORD definitions that have the same name but different fields. Such conflicts are detected when the programs are loaded. It is recommended that the programmer use a unique naming convention for RECORD definitions.

A RECORD type definition can be referred to in other programs if it is defined with the GLOBAL attribute and if it is IMPORTed in such programs by means of the IMPORT statement (for further details see [par. 3.2.2 TYPE declarations on page 63](#) and [par. 3.2.4.2 GLOBAL attribute and IMPORT statement on page 68](#))

The programmer can define a RECORD to have as many fields as needed, however, the maximum size for a record value is 65535 bytes.

Individual fields in a RECORD are referenced by separating the record variable name and the field name by a period. This is called field notation. For example:

```
rec_var.field_name := exp
```

All of the operations that are available for a particular data type can be performed on individual fields of that type. An entire RECORD can be used as an argument in a routine call or used in an assignment statement. When using records in assignment statements, both records must be of the same RECORD definition.

3.1.7 VECTOR

The VECTOR data type represents a quantity having both direction and magnitude. It consists of three REAL components. Vectors usually represent a location or direction in Cartesian space, with the components corresponding to x, y, z coordinates. [Fig. 3.3 - Representing Vectors](#) shows an example of a vector.

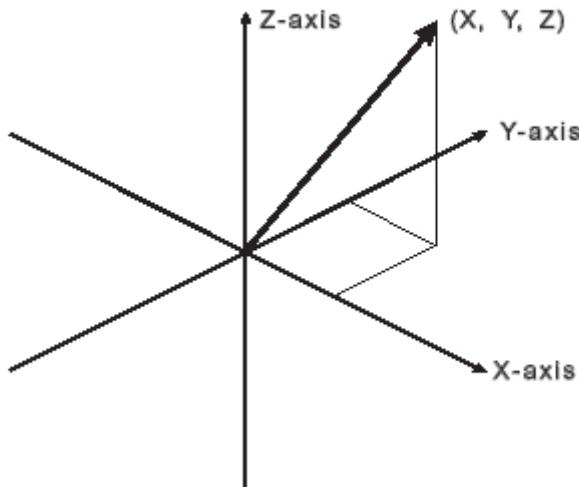
PDL2 can perform the following operations on VECTOR data:

- arithmetic (+, -);
- arithmetic (*, /) VECTOR-INTEGER, VECTOR-REAL, and vice-versa;
- relational (=, <>);
- vector (#, @).

Individual components of a VECTOR are referenced using field notation with the predefined fields X, Y, and Z. For example:

```
vec_var.X := 0.65
```

Fig. 3.3 - Representing Vectors



The VEC built-in routine also provides access to these components.

3.1.8 POSITION

The POSITION data type is used to describe the position of a Cartesian frame of reference with respect to another (called *starting frame*).

Generally a position is used to specify the final point for a MOVE statement that is the position to be reached by the end-of-arm tooling with respect to the user frame. POSITIONS also define the system frames of reference: for example the position of the base of the robot (\$BASE), the dimensions of the end-of-arm tooling (\$TOOL) and the user frame linked to the workpiece (\$UFRAME) (see [par. 3.1.8.1 Frames of reference on page 59](#) for a general definition of frames).

Note that the POSITION defines not only the location but also the orientation and, only for the final points, the configuration of the robot. Therefore POSITION data type consists of three REAL location components, three REAL orientation components, and a STRING that contains the configuration components.

The location components represents distances, measured in millimeters, along the x, y, z axes of the starting frame of reference. As with vectors, the x, y, z components can be referenced using the predefined fields x, y and z.

The orientation components represent three rotation angles, measured in degrees, called Euler angles. They allow to univocally define the final orientation of a frame of reference by applying to the starting frame three consecutive rotations. The first rotation (first Euler angle E1) is around the Z axis of the starting frame, the second is around the Y axis of the resulting frame (angle E2), the third is around the Z axis of the final frame (angle E3). The limits for the first and third Euler angle are from -180 to 180; the limits for the second are from 0 to 180 (always positive value). Euler angles can be referenced using the predefined constants A, E and R

[Fig. 3.4 - Euler Angles of Rotation](#) shows an example of a POSITION used to describe a final point for a MOVE statement.

When the POSITION is used to define a final point for a MOVE statement the configuration component is necessary to represent a unique set of the robot joint angles that bring the TCP on that position. The set of possible components is related to the family of robots. Note that the configuration string is automatically assigned by the system when teaching a final point so that generally they are not to be written explicitly in a program.

The configuration string can contain two type of information: the attitude flags and the turn flags.

The letters used as attitude flags in the configuration string are S, E, W, A and B. Some of these flags may be invalid on some robot arms. The entire set of components only apply to the SMART family of robots. The arm shoulder, elbow and wrist configuration are represented by the characters S, E and W. Here follows for example the description of the attitude flags for the SMART robots:

- S, if present in the configuration string, indicates that the robot must reach the final point with the WCP (Wrist Center Point) in the hinder space with respect to the plane defined by the first and second axes;
- E, if present, indicates that the WCP must be in the zone lying behind the extension of the second axis;
- W, if present, indicates that the robot must reach the final point with a negative value for the fifth axis.

The characters A and B represent the configuration flags for the wrist-joint motions (that are cartesian trajectories with \$ORNT_TYPE=WRIST_JNT). These flags are ignored when performing non-wrist-joint motions, just like the S, E and W flags are ignored during wrist-joint motions. The meaning of the attitude flags for a SMART robot follows:

- the character A, if present, indicates that the robot must reach the final point with the TCP (Tool Center Point) in the hinder space with respect to the plane defined by the second and third axes;
- the character B, if present, indicates that the robot must reach the final point with the TCP in the hinder space with respect to the plane defined by the first and second axes.

The turn flags are useful for robot axes that can rotate for more than one turn (multi-turn axes). For this type of robots the same position can be reached in different axis

configurations that differ for one or more turns (360 degrees). There are four turn flags called: T1, T2, T3, T4.

The syntax inside the configuration string is Ta:b, where 'a' represents the flag code (1 to 4) and 'b' represents the number of turns (-8 to +7).

The link between the flag name and the axis is shown in the following table:

Tab. 3.1 - Link between flags and axes for different robots

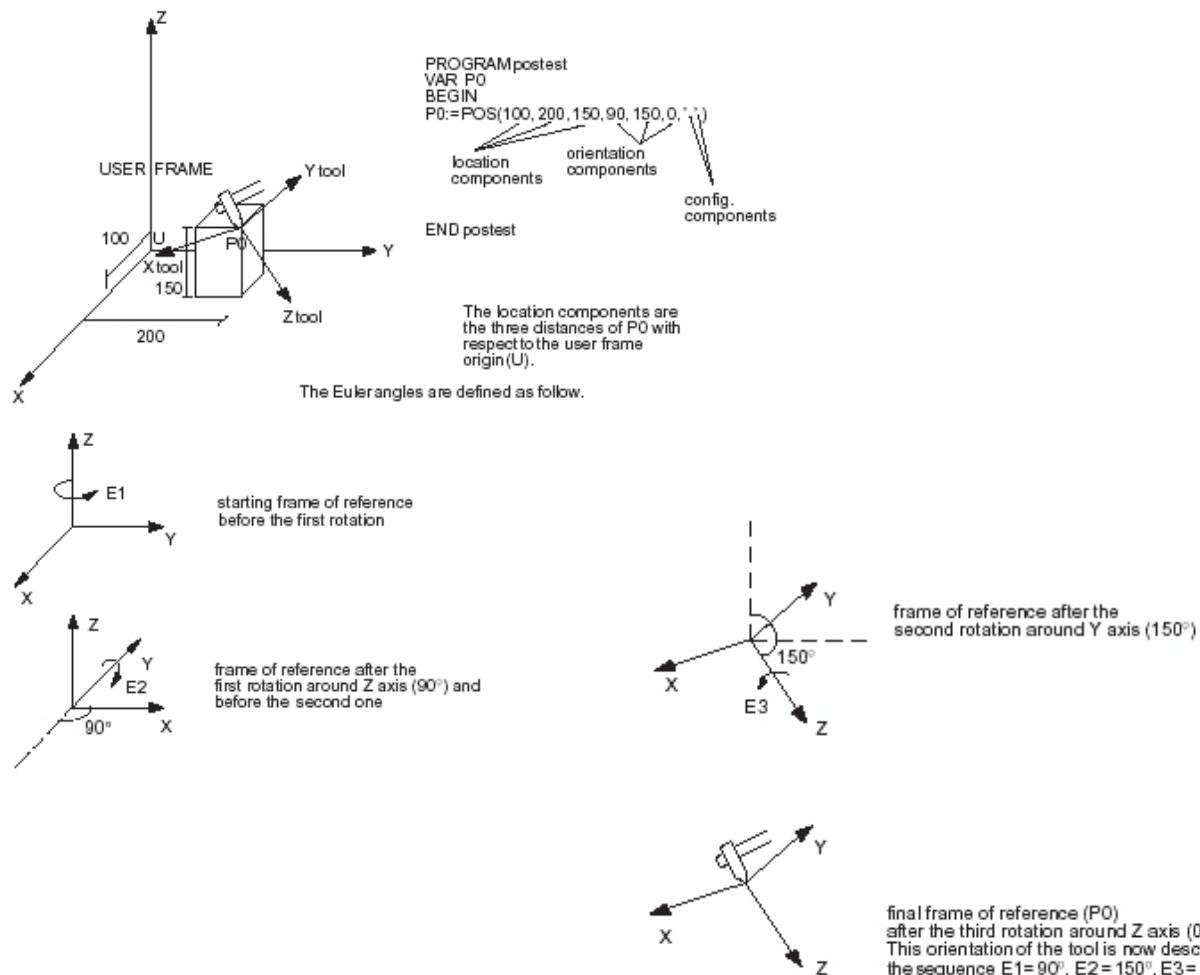
NORMAL CONFIGURATION	
Turn Flags	Robot Axis
T 1	ax 4
T 2	ax 6
T 3	ax 3
T 4	ax 5

Any combination of S, E, W, A, B and Ta:b can be used in the configuration string and in any order.

An example of valid configuration string follows.

S E W A T1:1 T2:-1 T3:2

Fig. 3.4 - Euler Angles of Rotation



PDL2 can perform the following operations on POSITION data:
 relative position (:) POSITION-POSITION, POSITION-VECTOR

The following example program shows how to declare and use a POSITION variable:

```

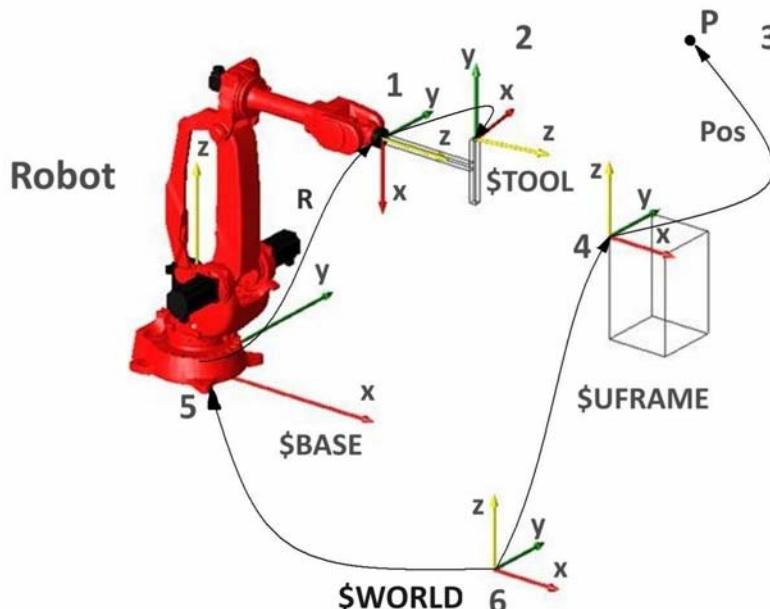
PROGRAM postest
VAR
    pos_var : POSITION
BEGIN
    pos_var := POS(294, 507, 1492, 13, 29, 16, )
END pos
    
```

PDL2 provides built-in routines to perform common POSITION manipulations, including accessing individual components of a position (refer to [BUILT-IN Routines List](#) chapter).

3.1.8.1 Frames of reference

Positions can be used to represent Cartesian frames of reference or the positions of objects relative to Cartesian frames of reference. In PDL2, positions are defined relative to a Cartesian frame of reference, as shown in the following [Fig. 3.5 - System Frames of Reference and Coordinate Transformation](#).

Fig. 3.5 - System Frames of Reference and Coordinate Transformation



1 - Flange Frame	2 - Tool Frame	3 - Taught Position
4 - User Frame	5 - Base Frame	6 - World Frame

The world frame is predefined for each arm. The programmer can define the base frame (\$BASE) as a position, relative to the world frame. The programmer also can define the end-of-arm tooling (\$TOOL) as a position, relative to the faceplate of the arm. \$UFRAME is a transformation used to describe the position of the workpiece with respect to the world.

Relative frames can be used to compensate for changes in the workcell, without having to reteach positional data. For example, \$BASE can be changed if the arm is relocated in the workcell or \$TOOL can be changed if the end-of-arm tooling changes. Relative

frames can also be assigned to parts, such as a car body. Positional data can then be taught relative to that part, for example, particular weld spots. If the position of the car body changes, only the frame needs to be retaught to correct all of the weld spots.

3.1.9 JOINTPOS

The JOINTPOS data type represents the actual arm joint positions, in degrees. One real component corresponds to each joint of the arm.

JOINTPOS data is used to position the end-of-arm tooling using a particular set of joint movements. Each real value is the actual distance a joint must move from its predefined “zero” position. Each JOINTPOS variable is associated with a particular arm and cannot be used with a different arm.

Individual components of a JOINTPOS, like ARRAY components, are referenced by index numbers.

For example:

```
PROGRAM jnttest
VAR
  real_var : REAL
  jointpos_var : JOINTPOS
BEGIN
  real_var := jointpos_var[5]
  jointpos_var[3] := real_exp
END jnttest
```

There are no operations for the entire JOINTPOS data type. PDL2 provides built-in routines to perform JOINTPOS manipulations (refer to [BUILT-IN Routines List](#) chapter).

3.1.10 XTNDPOS

The XTNDPOS data type represents an arm position that involves a greater number of axes than is included in the basic configuration of the robot. It is used for integrated motion of a group of axes, made up of a robot arm and some additional auxiliary axes, treated as a single unit in the system. For example, an XTNDPOS could be used to represent a robot mounted on a rail. The robot and rail would be treated as a single arm by the system. Each XTNDPOS variable is associated with a particular arm and cannot be used with a different arm.

The XTNDPOS data type is composed of a Cartesian position for the robot and an ARRAY of joint values for the remaining axes.

Individual components of an XTNDPOS are referenced using field notation with the predefined fields POS, a POSITION, and AUX, an ARRAY of REAL. For example:

```
PROGRAM auxaxis
VAR
  plxtn : XTNDPOS
BEGIN
  plxtn.POS := POS(294, 507, 1492, 13, 29, 16, )
  plxtn.AUX[1] := 100
  plxtn.AUX[2] := 150
END auxaxis
```

There are no operations for the entire XTNDPOS data type.

3.1.11 NODE

The NODE data type is similar to a RECORD in that it represents a collection of one or more data items grouped together using a single name. Each item of a node is called a field and can be of any PDL2 data type except SEMAPHORE, RECORD, NODE, or PATH.

The difference between a NODE and a RECORD is that a NODE can include a group of predefined node fields in addition to user-defined fields. The predefined node fields begin with a \$ and have a meaning known to the system identical to the corresponding predefined variable. The collection of predefined node fields contains a destination and description of a single motion segment (motion segments are described in [Chap.4. - Motion Control](#)).

Node data types are defined in the TYPE section of a program. A node type definition creates a user-defined data type that is available to the entire system. This means that it is possible to have conflicts between RECORD and NODE definitions that have the same name but different fields. Such conflicts are detected when the programs are loaded. It is recommended that the programmer use a unique naming convention for RECORD and NODE definitions.

Like for RECORD data types, a NODE type definition can be referred to in other programs if it is defined with the GLOBAL attribute and if it is IMPORTed in such programs by means of the IMPORT statement (for further details see [par. 3.2.2 TYPE declarations on page 63](#) and [par. 3.2.4.2 GLOBAL attribute and IMPORT statement on page 68](#))

The programmer can define a node to have as many fields as needed, however, the maximum size for a node value is 65535 bytes.

Individual fields in a node are referenced by separating the node variable name and the field name by a period. This is called field notation. For example:

```
node_var.$PRED_FIELD_NAME := exp -- ref. predefined node field
node_var.field_name := exp -- ref. user-defined node field
```

All of the operations that are available for a particular data type can be performed on individual fields of that type. An entire node can be used as an argument in a routine call, used in an assignment statement, or used as the destination in a MOVE statement. When using nodes in assignment statements, both nodes must be of the same node definition.

3.1.12 PATH

The PATH data type represents a sequence of nodes to be interpreted in a single motion. The PATH data type is a predefined record containing the fields NODE, FRM_TBL, and COND_TBL.

The NODE field is an ARRAY of nodes representing the sequence of nodes. It is dynamic in length which means nodes can be added and deleted from the table during execution. The maximum number of nodes in a path is 65535 but the amount of memory on the system may not permit this many nodes.

The structure of the nodes in the NODE array is determined by the user-defined node definition declared in the TYPE section of the program. Each node contains a destination and description for a single motion segment (motion is described in [Motion Control chapter](#)). This provides the programmer with the ability to customize the node definitions for different applications. Please note that it is possible to have different paths use different node definitions, however, all nodes within the same path will have the

same node definition.

The node type is available to the entire system. This means that it is possible to have conflicts between node types that have the same name but different field definitions. It is recommended that the programmer use a unique naming convention for node definitions in order to avoid such conflicts.

Individual nodes of a path are referenced by indexing the NODE field of the path variable.

For example:

```
path_var.NODE[3] := path_var.NODE[5]
path_var.NODE[4] := node_var
```

Individual fields in a path node are referenced using field notation. For example:

```
path_var.NODE[3].field_name := exp
```

All of the operations that are available for a particular data type can be performed on individual fields of that type.

The FRM_TBL field is an ARRAY of POSITIONS representing reference and/or tool frames to be used during the processing of the path nodes. The FRM_TBL array contains 7 elements. The usage of the FRM_TBL field is described in the PATH Motion section of [Chap.4. - Motion Control](#).

The COND_TBL field is an ARRAY of INTEGERS representing condition handler numbers to be used during the processing of the path nodes. The COND_TBL array contains 32 elements. The usage of the COND_TBL field is described in the PATH Motion section of [Chap.4. - Motion Control](#).

There are no operations for the entire PATH type. An entire path can be used in the MOVE ALONG statement or as an argument in a routine call. PDL2 provides built-in routines to insert, delete, and append nodes to a path. Additional built-ins are provided to obtain the number of nodes in a path and assign/obtain a node identifier (refer to [Chap.4. - Motion Control](#)).

3.1.13 SEMAPHORE

The SEMAPHORE data type represents an aid for synchronization when there are multiple programs running that share the same resources. The SEMAPHORE, or an array of SEMAPHOREs is used to provide mutual exclusion of a resource so that separate programs cannot act on that resource at the same time.

There are no operations for the SEMAPHORE data type, but the following statements use SEMAPHORE variables:

- [WAIT Statement](#);
- [SIGNAL Statement](#).

[Execution Control](#) chapter provides more information about using SEMAPHOREs.

3.2 Declarations

This section describes

- [CONSTANT declarations](#),
- [VARIABLE declarations](#),
- [TYPE declarations](#).
- [Shared types, variables and routines](#)

3.2.1 CONSTANT declarations

Constants are declared in the CONST section of a program (see [par. 10.12 CONST Statement on page 189](#)).

A CONSTANT declaration establishes a constant identifier with two attributes: a name and an unchanging value.

The data type of the CONSTANT is understood by its assigned value, which can be an INTEGER, a REAL, a BOOLEAN, or a STRING. Within the program, the identifier can be used in place of the value.

The syntax for declaring a constant is as follows:

```
name = || literal | predef_const_id ||
```

Constant declarations appear in a constant declaration section, following the reserved word CONST. For example:

```
CONST
  num_parts = 4
  max_angle = 180.0
  part_mask = 0xF3
  test_flag = TRUE
  error = 'An error has occurred.'
```

PDL2 provides predefined constants for representing commonly used values. These predefined constants are listed in [Tab. 2.3 - Predefined Constants of Introduction to PDL2 chapter](#).

3.2.2 TYPE declarations

RECORD and NODE definitions are declared in the TYPE declaration section of a program (see [par. 10.46 TYPE Statement on page 218](#)). A type declaration establishes a type identifier with two attributes; a name and a RECORD or NODE definition.

The syntax for declaring a RECORD definition is as follows:

```
type_name = RECORD <GLOBAL>
  <fld_name <, fld_name>... : data_type>...
ENDRECORD
```

The syntax for declaring a node definition is as follows:

```
type_name = NODEDEF <GLOBAL>
  <predefined_name <, predefined_name>... <NOTEACH> >...
  <fld_name <, fld_name>... : data_type <NOTEACH> >...
ENDNODEDEF
```

The *type_name* is the identifier for the user-defined type. This is used in variable and parameter declarations to indicate a RECORD or NODE type.

GLOBAL attribute means that current user-defined type is declared to be public for use by other programs. See [par. 3.2.4 Shared types, variables and routines on page 67](#) for full details.

The *fld_name* identifiers indicate the user-defined fields of the RECORD or NODE type.

The *predefined_name* identifiers in a node definition indicate the set of motion segment characteristics contained in each node. As indicated above, the predefined node fields must be specified before the user-defined node fields.

Each predefined_name can be one of the following:

\$CNFG_CARE	\$COND_MASK	\$COND_MASK_BACK
\$JNT_MTURN	\$LIN_SPD	\$MAIN_JNTP
\$MAIN_POS	\$MAIN_XTND	\$MOVE_TYPE
\$ORNT_TYPE	\$ROT_SPD	\$SEG_DATA
\$SEG_FLY	\$SEG_FLY_DIST	\$SEG_FLY_PER
\$SEG_FLY_TYPE	\$SEG_FLY_TRAJ	\$SEG_OVR
\$SEG_REF_IDX	\$SEG_STRESS_PER	\$SEG_TERM_TYPE
\$SEG_TOL	\$SEG_TOOL_IDX	\$SEG_WAIT
\$SING_CARE	\$SPD_OPT	\$TURN_CARE
\$WEAVE_NUM	\$WEAVE_TYPE	

The meaning of each predefined node field is identical to the predefined variable having the same name. These are described in the [Motion Control chapter](#) and [Predefined Variables List chapter](#).

The \$MAIN_POS, \$MAIN_JNTP, and \$MAIN_XTND fields indicate the main destination of a motion segment. A node definition can include only one of the \$MAIN_ predefined fields. The particular one chosen indicates the data type of the main destination.

If a predefined node field is used in the program and not included in the node definition, the program editor will automatically insert that field in the declaration. This is called an implicit declaration.

The NOTEACH clause is used to indicate that the fields declared in that declaration should not be permitted to be changed while a path is being modified in the teaching environment (MEMORY TEACH Command). (Refer to the *C5G Control Unit Use* manual for more information on the teaching environment.)

Type declarations appear in a type declaration section, following the reserved word TYPE. For example:

```

TYPE
  ddd_part = RECORD GLOBAL -- declared to be IMPORTable by
                      -- other programs
    name : STRING[15]
    count : INTEGER
    params : ARRAY[5] OF REAL
  ENDRECORD

  lapm_pth1 = NODEDEF
    $MAIN_POS, $SEG_TERM_TYPE
    $MOVE_TYPE
    $SEG_WAIT NOTEACH
    weld_sch : ARRAY[8] OF REAL
    gun_on : BOOLEAN
  ENDNODEDEF

```

The type declaration is just a definition for a new data type. This new data type can be used for variable and parameter declarations.

3.2.3 VARIABLE declarations

Variables are declared in the variable declaration section of a program (see [par. 10.49 VAR Statement on page 220](#)).

A VARIABLE declaration establishes a variable identifier with two attributes: a name and a data type. Within the program, the variable can be assigned any value of the declared data type.

The syntax for declaring a variable is as follows:

```
name <, name>... : data_type <var_options>
```

The valid data types (*data_type*) and their syntax are as follows:

```
INTEGER
REAL
BOOLEAN
STRING [length]
ARRAY [rows <, columns>] OF item_type (see par. 3.1.5 ARRAY on page 54)
record_type
node_type
VECTOR
POSITION
JOINTPOS <FOR ARM[number]>
XTNDPOS <FOR ARM[number]>
PATH OF node_type
SEMAPHORE
```

The possible values and ranges for length, rows, columns, and item_type are explained in [par. 3.1 Data Types on page 51](#) of current chapter.

The length of a STRING or the size(s) of an ARRAY can be specified with an * instead of an actual value. If the * notation is used with a two-dimensional ARRAY, then two * must be used (*, *). The * notation is used when importing STRING or ARRAY variables from another program. The program owning the variable should specify the actual size of the variable while the importing program(s) use the * notation (refer to the example in the Shared Variables and Routines section of this chapter). If the variable is accessed before an actual size for it has been determined, an error will occur. The actual size is determined when a program or variable file specifying the actual size is loaded. The importing programs can obtain the actual size using built-in routines (refer to [BUILT-IN Routines List](#) chapter).

The * notation is not allowed in local routine variables or field definitions.

Arm designations are set up at the system level by associating an arm number with a group of axes. They are not required for single arm systems (refer to the *C5G Control Unit Use* manual).

If an arm designation is not specified in the declaration of a JOINTPOS or XTNDPOS, the program attribute PROG_ARM is used. If PROG_ARM is not specified in the program, the default arm (\$DFT_ARM) is used.

The valid *var_options* are as follows:

- **EXPORTED FROM** clause and **GLOBAL** attribute
See [par. 3.2.4 Shared types, variables and routines on page 67](#) for full details.
- initial_value

The initial_value clause is permitted on both program and routine variable declarations. However, it is only valid when the data type is INTEGER, REAL, BOOLEAN, or STRING. This option specifies an initial value for all variables declared in that declaration statement. For program variables, the initial value is given to the variables at the beginning of each program activation and for routine variables at the beginning of each routine call (refer to [Routines](#) chapter for more information on routine variables.)

The initial value can be a literal, a predefined constant identifier, or a user-defined constant identifier. The data type of the initial value must match the data type of the variable declaration with the exception that an INTEGER literal may be used as the initial value in a REAL variable declaration.

- **NOSAVE** attribute

The values of program variables may be saved in a variable file so that they can be used the next time the program is activated. (Refer to the *C5G Control Unit Use* manual for more information on variable files) The NOSAVE clause is used to indicate that the variables declared in that declaration should not be saved to or loaded from the variable file. This option is only permitted on program variable declarations and applies to all variables declared in the declaration.

The NOSAVE clause is automatically added to all SEMAPHORE variable declarations since they are never permitted in a variable file.

The NOSAVE option should be used on all program variable declarations that include the initial value clause. If not specified, the program editor will give the user a warning. The reason is that initialized variables will be given the initial value each time the program is activated which, in effect, overrides any value loaded from the variable file.

- **CONST** attribute

The CONST attribute can be applied to a variable to mean that it has privileged write access and any read access: non-privileged users can neither set a value to such a variable, nor pass it by reference to a routine. The only way to set a value to it is to load it from a file by means of the **ML/V** Command (Memory Load / Variables - for further details see [System Commands](#) chapter in [C5G Control Unit Use](#) manual).

- **NODATA** attribute

The NODATA attribute allows to avoid displaying the corresponding variable on the TP DATA Page (see also [par. 16.2 User table creation from DATA environment on page 558](#)).

Variable declarations appear in a variable declaration section, following the reserved word VAR. For example:

```
VAR
  count, total : INTEGER (0) NOSAVE
  timing : INTEGER (4500)
  angle, distance : REAL
  job_complete, flag_check, flag_1, flag_2 : BOOLEAN
  error_msg : STRING[30] GLOBAL
  menu_choices : ARRAY[4] OF STRING[30]
  matrix : ARRAY[2, 10] OF INTEGER
  offset : VECTOR
  part_rec : ddd_part
  pickup, perch : POSITION
```

```

safety_pos : JOINTPOS FOR ARM[2]
door_frame : XTNDPOS FOR ARM[3]
weld_node : lapm_pth1
weld_pth : PATH OF lapm_pth1
work_area : SEMAPHORE NOSAVE
default_part : INTEGER (OxFF) NOSAVE
    
```

If a programmer uses an undeclared variable identifier in the executable section of a program, the program editor automatically adds a declaration for that variable. This is called an implicit declaration. The variable must be used in a way that implies its data type or an error will occur. The new variable is added to an existing variable declaration of the same data type if one exists containing less than 5 variables. If one does not exist, a new declaration statement is added. Undeclared variables cannot be used in the executable section of a routine or an error will occur.

PDL2 provides predefined variables for data related to motion, input/output devices, and other system data. [Predefined Variables List](#) chapter describes these variables.

3.2.4 Shared types, variables and routines

- [EXPORTED FROM clause](#)
- [GLOBAL attribute and IMPORT statement](#)

3.2.4.1 EXPORTED FROM clause

Variables and routines can be declared to be owned by another program or to be public for use by other programs, by means of the optional EXPORTED FROM clause. This allows programs to share variables and routines. The EXPORTED FROM clause can be used in any program variable or routine declaration ([Chap.7. - Routines](#) explains routine declarations).

The syntax for declaring a shared variable or routine is as follows:

```
EXPORTED FROM prog_name
```

prog_name indicates the name of the program owning the specified variable or routine. If this is the same name as the program in which this statement resides, the variable or routine can be accessed by other programs.

If *prog_name* is different from the name of the program in which this statement resides, the variable or routine is owned by *prog_name*. In this case, the program that owns the item must also export it or an error will occur when the program is loaded.

The following example shows how to use the EXPORTED FROM clause for shared variables.

<pre> PROGRAM a VAR x : INTEGER EXPORTED FROM a y : REAL EXPORTED FROM b ary: ARRAY[5] OF REAL EXPORTED FROM a BEGIN . . . END a </pre>	<pre> PROGRAM b VAR y : REAL EXPORTED FROM b x : INTEGER EXPORTED FROM a ary: ARRAY[*] OF REAL EXPORTED FROM a BEGIN . . . END b </pre>
---	---

```
-- x and ary are open for other programs      -- y is open for other programs to use
to use
-- y is owned by program b                  -- x and ary are owned by program a
```



The EXPORTED FROM clause does not apply to routine local variables. In addition, the initial value clause cannot be included when the EXPORTED FROM clause specifies a program name different from the name of the program in which the statement resides.

```
PROGRAM a
ROUTINE rout_a(x : INTEGER) EXPORTED FROM a
ROUTINE rout_b(x, y : INTEGER) EXPORTED FROM b

ROUTINE rout_a(x:INTEGER)
BEGIN
  .
  .
  END rout_a
BEGIN
  .
  .
  END a
-- rout_a is open for other programs to use
-- rout_b is owned by program b
```

3.2.4.2 GLOBAL attribute and IMPORT statement

User-defined types, variables and routines can be declared to be IMPORTed by other programs or to be public for use by other programs, by means of the optional IMPORT statement and the GLOBAL attribute.

The GLOBAL attribute allows programs to share typedefs, variables and routines. This means that other programs without explicitly declaring them, can IMPORT these items.

- The syntax for declaring a **user-defined type** to be public, is as follows:

```
rec_name = RECORD GLOBAL
node_name = NODEDEF GLOBAL
```

rec_name and *node_name* indicate the name of the declared type. In such a way, it can be easily used by other programs.

- The syntax for declaring a **variable** to be public, is as follows:

```
variable_name GLOBAL
```

variable_name indicates the name of the declared variable. In such a way, the declared variable can be accessed by other programs.

- The syntax for declaring a **routine** to be public, is as follows:

```
routine_name EXPORTED FROM progr_name GLOBAL
```

routine_name indicates the name of the declared routine. In such a way, the declared routine can be called by other programs.

progr_name indicates the name of the current program, which owns the routine called *routine_name* and declares it to be public.



GLOBAL attribute doesn't apply to routine local variables.

The IMPORT statement must be used to import ANY GLOBAL types, variables and/or routines from another program (see [par. 10.28 IMPORT Statement on page 201](#)).

The syntax for importing them is as follows:

```
IMPORT 'prog_name'
```

prog_name is the name of the program which owns the types, variables and routines to be imported. They all are imported in the current program without explicitly declaring them.



NOTE that PDL2 Programs using the IMPORT statement cannot be opened in Program Edit environment.

The following example shows how to use the IMPORT clause and the GLOBAL attribute, for shared variables.

PROGRAM a

```
IMPORT 'b'      -- causes just y to be imported from program b
VAR
    x : INTEGER GLOBAL          -- declared to be public
    ary : ARRAY [5] OF REAL GLOBAL -- declared to be public
ROUTINE rout(x:REAL) EXPORTED FROM a GLOBAL -- declared to be
                                              -- public
BEGIN
    .
    .
    .
END rout
BEGIN
    .
    .
    .
END a
```

PROGRAM b

```
IMPORT 'a' -- causes x, ary and rout to be imported from program a
VAR
    y : REAL GLOBAL -- declared to be public
    i : INTEGER     -- declared to be local to program b (not public)
BEGIN
    .
    .
    .
END b
```

3.3 Expressions

Expressions are combinations of any number of constants, variables, function routines, and literals joined with operators to represent a value. For example:

```
count + 1           -- arithmetic
VEC(a, b, c) * 2   -- arithmetic
```

```

count >= total           -- relational
flag_1 AND flag_2        -- BOOLEAN
  
```

An expression has both a type and a value. The type is determined by the operators and operands used to form it. [Tab. 3.2 - Operation Result Types](#) shows which operand data types are allowed for which operators and what data types result. The resulting value can be assigned to a variable of the same type using an assignment statement.

In [Tab. 3.2 - Operation Result Types](#), the following abbreviations are used:

I	INTEGER	V	VECTOR
R	REAL	P	POSITION
B	BOOLEAN		

Tab. 3.2 - Operation Result Types

Operators:	+, -	*	/	DIV, MOD	=,<> <,<= >,>=	AND, NOT, OR, XOR	#	@	:	ROR ROL SHR SHL	**	+= -=
Operand Types:												
INTEGER	I	I	R	I	B	I	—	—	—	I	I	I
REAL	R	R	R	—	B	—	—	—	—	—	—	—
INTEGER-REAL	R	R	R	—	B	—	—	—	—	—	—	—
REAL-INTEGER	R	R	R	—	B	—	—	—	—	—	R	—
BOOLEAN	—	—	—	—	B	B	—	—	—	—	—	—
STRING	—	—	—	—	B	—	—	—	—	—	—	—
INTEGER-VECTOR	—	V	V	—	—	—	—	—	—	—	—	—
REAL-VECTOR	—	V	V	—	—	—	—	—	—	—	—	—
VECTOR	V	—	—	—	B ⁽¹⁾	—	V	R	—	—	—	—
POSITION	—	—	—	—	—	—	—	—	P	—	—	—
POSITION-VECTOR	—	—	—	—	—	—	—	—	V	—	—	—

(1) Only the operators = and <> may be used to compare VECTOR values.

The following operation types are available:

- [Arithmetic Operations](#)
- [Relational Operations](#)
- [Logical Operations](#)
- [Bitwise Operations](#)
- [VECTOR Operations](#)
- [POSITION Operations](#)

3.3.1 Arithmetic Operations

Arithmetic operators perform standard arithmetic operations on INTEGER, REAL, or VECTOR operands. The arithmetic operators are as follows:

- + INTEGER, REAL or VECTOR addition

- INTEGER, REAL or VECTOR subtraction
- * INTEGER, REAL or VECTOR multiplication
- DIV** INTEGER division (fractional results truncated)
- / REAL or VECTOR division
- MOD** INTEGER modulus (remainder)
- ** INTEGER or REAL exponentiation
- + =** INTEGER increment
- =** INTEGER decrement

VECTOR addition and subtraction require VECTOR operands and produce a VECTOR result whose components are the sum or difference of the corresponding elements of the operands.

VECTOR scalar multiplication and division require a VECTOR operand and an INTEGER or REAL operand and produce results obtained by performing the operation on each element in the vector (an INTEGER operand is treated as a REAL). If an INTEGER or REAL number is divided by the VECTOR, that value is multiplied by the reciprocal of each element of the VECTOR.

PDL2 provides built-in routines to perform common mathematical manipulations, including rounding and truncating, trigonometric functions, and square roots (refer to [BUILT-IN Routines List](#) chapter).

3.3.2 Relational Operations

Relational operators compare two operands and determine a BOOLEAN value based on whether the relational expression is TRUE or FALSE. The relational operators are as follows:

- < less than
- > greater than
- = equal
- <= less than or equal
- >= greater than or equal
- <> not equal

Relational operations can be performed on INTEGER, REAL, BOOLEAN (= or <> only), STRING, and VECTOR (= or <> only) values.

3.3.3 Logical Operations

When used with BOOLEAN values, the BOOLEAN operators work as logical operators to generate a BOOLEAN result. All of the BOOLEAN operators require two operands except NOT, which only requires one operand.

- AND** TRUE if both operands are TRUE
- OR** TRUE if at least one of the operands is TRUE

XOR TRUE if only one operand is TRUE

NOT inverse of the BOOLEAN operand

3.3.4 Bitwise Operations

Bitwise operations perform a specific operation on the bits of an INTEGER value. There are three different ways that bitwise operations can be performed: by using BOOLEAN operators, rotate or shift operators, and built-in procedures (BIT_TEST, BIT_SET, and BIT_CLEAR).

The rotate and shift operators require INTEGER operands and produce an INTEGER result. The left operand is the value to be shifted or rotated and the right operand specifies the number of bits to shift or rotate. The shift operators perform an arithmetic shift which causes the shifted bits to be discarded, zeros to be shifted into the vacated slots on a shift left, and a copy of the signed bit (bit 32) from the original value to be shifted into the vacated positions on a shift right. The rotate operators cause the shifted bit(s) to be wrapped around to the opposite end of the value. Fig. 3.6 and Fig. 3.7 show an example of the shift left and rotate left operations. Fig. 3.8 shows an example of a shift right instruction. Fig. 3.6 shows the result of the following PDL2 statement:

```
x := -122704229 SHL 1
```

Fig. 3.6 - Shift Left Operator

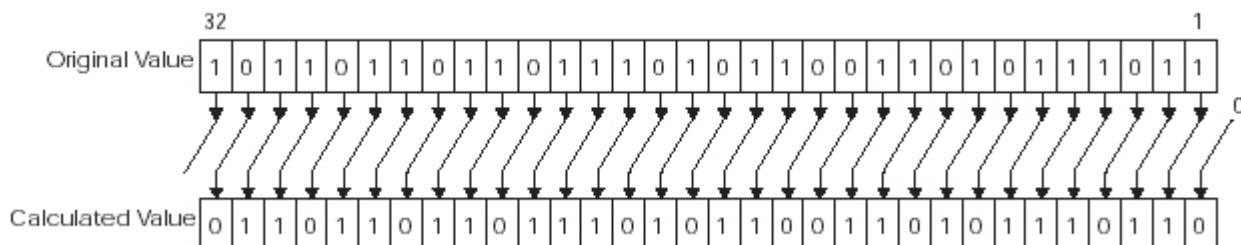


Fig. 3.7 shows the result of the following rotate operation:

```
x := -122704229 ROL 1
```



NOTE that the Shift Left operation might cause the variable to become UNINIT: at run time, an INTEGER PDL2 variable will assume the value of UNINIT (uninitialized) if it becomes greater than MAXINT.

Fig. 3.7 - Rotate Left Operator

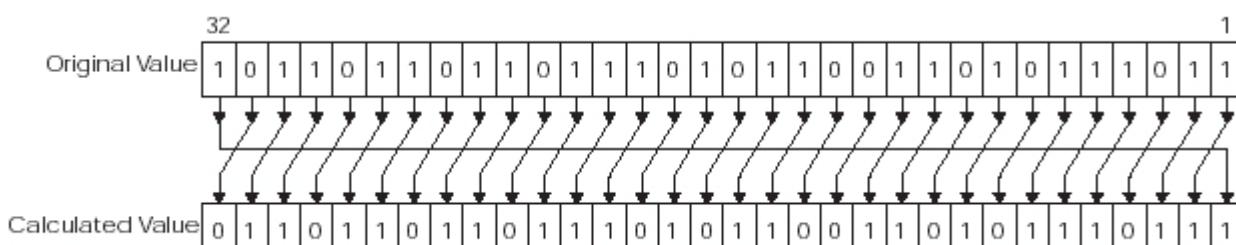
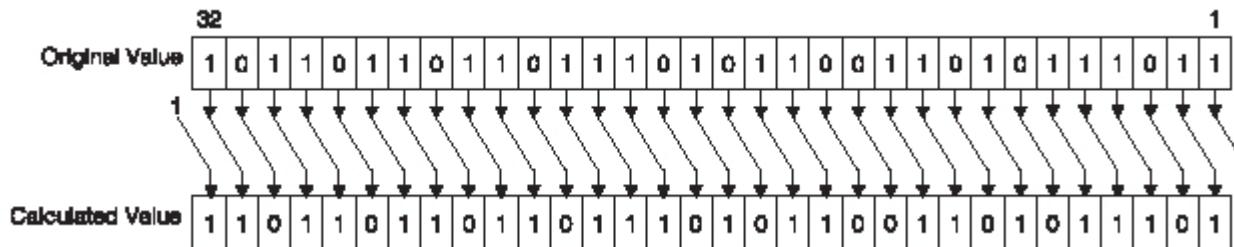


Fig. 3.8 shows the result of the shift right operation:

`x := -122704229 SHR 1`

Fig. 3.8 - Shift Right Operator



The operations performed by BOOLEAN, rotation, and shift operators are listed below. Refer to [BUILT-IN Routines List](#) chapter for explanations of BIT_TEST, BIT_SET, and BIT_CLEAR.

AND	ith bit is 1 if ith bit of both operands is also 1
OR	ith bit is 1 if ith bit of either, or both, operands is 1
XOR	ith bit is 1 if ith bit of only one operand is 1
NOT	ith bit is the reverse of ith bit of the operand
ROR	INTEGER rotate right
ROL	INTEGER rotate left
SHR	INTEGER shift right
SHL	INTEGER shift left

3.3.5 VECTOR Operations

Special VECTOR operations include cross product and inner product operations.

The cross product operator (#) results in a VECTOR normal to the VECTOR operands. The magnitude is equivalent to the product of the magnitude of the operands and the sine of the angle between the operands. The direction is determined by the right hand rule. For example:

`VEC(2, 5, 7) # VEC(3, 9, 1) = VEC(-58, 19, 3)`

The inner product operator (@) results in a REAL value that is the sum of the products of the corresponding elements of the two VECTOR operands. For example:

`VEC(7.0, 6.5, 13.4) @ VEC(1.3, 5.2, 0.0) = 42.9`

3.3.6 POSITION Operations

The relative position operator (:) performs a special position operation. It can be used with two POSITION operands or a POSITION operand and a VECTOR operand. This is implemented by converting the position operand into a matrix, performing the operation, and converting the resulting matrix back into standard position format. In the case of a POSITION and VECTOR operation, the result is a vector.

Two POSITION operands result in a POSITION equivalent to the right hand operand, but relative to the coordinate frame of the left hand operand. For example:

```
POS(10, 20, 30, 0, 30, 0, ) : POS(10, 20, 30, 45, 0, 0, ) =
POS(33.66, 40, 50.981, 0, 30, 65, )
```

A POSITION and a VECTOR operand result in a VECTOR equivalent to the VECTOR operand, but relative to the POSITION operand. For example:

```
POS(10, 20, 30, 0, 30, 0, ) : VEC(10, 0, 0) = VEC(18.660, 20, 25)
```

When the two operands are both POSITIONS, the relative position operator (:) assigns the resulting OR of Shoulder, Elbow, Wrist flags included in the configurations of the two POSITION operands, to the resulting POSITION configuration.



It is recommended to always verify whether or not the resulting configuration is the expected one.

If not, use either [POS_SET_CNFG Built-In Procedure](#) or an [Assignment Statement](#) to the configuration field of the resulting POSITION, depending on the User's current needs.

Examples:

```
pos_set_cnfg(p3, 'S A B W')
pos3.cnfg := pos2.cnfg.
```



NOTE THAT the relative position operator (:), applied to two POSITION operands, does not handle Multiturn flags.

Three PDL2 program follow showing some examples of using the relative position operator:

```
PROGRAM monit_1 NOHOLD
VAR p1, p2, p3 : POSITION
BEGIN
  p1 := ARM_POS(1)
  p2 := ARM_POS(2)
  p3 := p1 : p2
  WRITE LUN_CRT (p1, NL, p2, NL, p3, NL, NL)
END monit_1
```

The output results are:

```
< -779.348, -691.505, -680.714, 136.195, 164.805, 0.151,>
< -1671.599, -51.389, 1315.723, 174.888, 153.262, -0.170, S A B>
< -2153.719, 704.028, -1512.339, 147.286, 11.669, 173.396, S A B>
```

The resulting configuration is the OR operation between the two operands configurations.

```

PROGRAM monit_2 NOHOLD
VAR p1, p2, p3 : POSITION
BEGIN
    p1 := ARM_POS(1)
    POS_SET_CNFG(p1, 'w')
    p2 := ARM_POS(2)
    p3 := p1 : p2
    WRITE LUN_CRT (p1, NL, p2, NL, p3, NL, NL)
END monit_2

```

The output results are:

< -779.348, -691.505, -680.714, 136.195, 164.805, 0.151, W >
 < -1671.599, -51.389, 1315.723, 174.888, 153.262, -0.170, S A B >
 < -2153.719, 704.028, -1512.339, 147.286, 11.669, 173.396, S W A B >

The resulting configuration is the OR operation between the two operands configurations.

```

PROGRAM monit_3 NOHOLD
VAR p1, p2, p3 : POSITION
BEGIN
    p1 := ARM_POS(1)
    POS_SET_CNFG(p1, 'w T1:1')
    p2 := ARM_POS(2)
    p3 := p1 : p2
    WRITE LUN_CRT (p1, NL, p2, NL, p3, NL, NL)
END monit_3

```

The output results are:

< -779.348, -691.505, -680.714, 136.195, 164.805, 0.151, W T1:1 >
 < -1671.599, -51.389, 1315.723, 174.888, 153.262, -0.170, S A B >
 < -2153.719, 704.028, -1512.339, 147.286, 11.669, 173.396, S W A B >

The multiturn flag has not been considered in the resulting configuration.

3.3.7 Data Type conversion

In PDL2 there is no implicit data type conversion. However, an INTEGER value can be used as a REAL value. PDL2 provides built-in routines to perform type conversions between some of the other data types (refer to [BUILT-IN Routines List](#) chapter.)

3.3.8 Operator precedence

Operators are evaluated in the order shown in [Tab. 3.3 - Operator Precedence](#). Precedence works left to right in a statement for operators that are at the same precedence level.

Tab. 3.3 - Operator Precedence

Priority	Operators
1	array []; field .
2	()
3	unary +; unary-; NOT

Tab. 3.3 - Operator Precedence (Continued)

Priority	Operators
4	**, vector @, #; position :
5	*, /, AND, MOD, DIV
6	+, -, OR, XOR
7	ROR, ROL, SHR, SHL
8	=, <>, <, <=, >, >=
9	assignment :=, increment +=, decrement -=



If parentheses are not used in an expression, the user should be aware of the exact order in which the operators are performed. In addition, those parentheses that are not required in the expression will automatically be removed by the system. For example, the first IF statement below will actually produce the following IF statement because the parentheses do not override normal operator precedence

```
IF ($DIN[5] AND $DIN[6]) = FALSE THEN
. . .
ENDIF

IF $DIN[5] AND $DIN[6] = FALSE THEN
. . .
ENDIF
```

If the FALSE test of \$DIN[6] is to be ANDed with \$DIN[5], parentheses must be used to override operator precedence:

```
IF $DIN[5] AND ($DIN[6] = FALSE) THEN
. . .
ENDIF
```

3.4 Assignment Statement

The ASSIGNMENT statement (:=) specifies a new value of a variable, using the result of an evaluated expression. The value resulting from the expression must be of the same data type as the variable.

Arm numbers for JOINTPOS and XTNDPOS variables must match and the dimension and size of arrays must match.

The syntax for an assignment is as follows:

```
variable := expression
```

Examples of assignment statements:

```
count := count + 1
offset := VEC(a, b, c) * 2
menu_choices[7] := 7. Return to previous menu
part_rec.params[1] := 3.14
part_mask := 0xE4
weld_pth.NODE[3].$SEG_WAIT := FALSE
```

```
$MOVE_TYPE := LINEAR
```

The PATH and SEMAPHORE data types cannot be assigned values.

If the same value is assigned to more than one user-defined variable of the same data type, multiple assignments can be put on the same line. System variables, predefined node fields and path nodes are not allowed. For example:

```
count1 := count2 := count3 := count + 1
offset1 := offset2 := VEC (a, b, c)
```

Multiple assignment on the same line is not allowed in condition actions.

3.5 Typecasting

This PDL2 feature involves INTEGER, BOOLEAN and REAL variables and values. It is a very specific feature and it is used in very particular applications.

The case in which only INTEGER and REAL data types are involved differs from the one in which also BOOLEAN data type is used. In both cases typecasting is expressed by specifying in round brackets the data type to apply to the casting operation.

In case of typecasting between INTEGER and REAL data types, the variable or value associated to the casting operation is read in the Controller memory and the bit pattern that forms this value is considered; this bit pattern is then assigned to the destination of the assignment or compared with the other operator of the expression.

Then used in assignments, casting can only be specified on the right hand side. When used in relational expressions, it can be specified in any side. This feature is allowed in program statements, condition actions and condition expressions. INTEGER variables, ports or values, and REAL variables or values are used.

Consider for example the numbers 0x40600000 and 0x3f9999A that are the hexadecimal representation of the bit pattern of 3.5 and 1.2 numbers in the C3G memory.

```
-- assign 0x40600000 to int_var
int_var := (INTEGER)3.5
-- assing 1.2 to real_var
real_var := (REAL)0x3f9999A
int_var := (INTEGER)(real_var + 3.4)

CONDITION[5] :
-- if real_var values 5.5 and int_var 0x3f9999A
-- the condition will trigger
WHEN (INTEGER)real_var> int_var DO
    real_var := (REAL)3
WHEN $AOUT[13] < (INTEGER) real_var DO
    int_var := (INTEGER)5.6
WHEN real_var > (REAL) int_var DO
    $AOUT[7] := (INTEGER)5.6
WHEN real_var > (REAL) $AOUT[4] DO
    int_var := (INTEGER) real_var
ENDCONDITION
```

In case of typecasting between INTEGER and BOOLEAN or REAL and BOOLEAN data types, the typecasting consists in assigning the value of 1 or 1.0 to the destination INTEGER or REAL variable respectively if the BOOLEAN variable (or port or value) is

TRUE, 0 otherwise.

This aspect of typecasting is not allowed in conditions handlers.
For example:

```
int_var := (INTEGER) bool_var
int_var := (INTEGER) $DOUT[5]
real_var := (REAL) bool_var
real_var := (REAL) $FDOOUT[6]
```

4. MOTION CONTROL

This chapter describes the PDL2 statements that control arm motion and direct hand operations.

Information are supplied about the following topics:

- MOVE Statement
- Motion along a PATH
- Stopping and Restarting motions
- ATTACH and DETACH Statements
- HAND Statements

4.1 MOVE Statement

The MOVE statement initiates arm motion. Different clauses and options allow for many different kinds of motion.

Information are supplied about the following topics:

- ARM Clause
- TRAJECTORY Clause
- DESTINATION Clause
- Optional Clauses
- SYNCMOVE Clause
- Continuous motion (MOVEFLY)
- Timing and Synchronization considerations
- ARM Clause
- NODE RANGE Clause
- Optional Clauses
- Continuous Motion (MOVEFLY)
- CANCEL MOTION Statements
- LOCK, UNLOCK, and RESUME Statements
- SIGNAL SEGMENT Statement
- HOLD Statement

The syntax of the MOVE statement is as follows:

```
MOVE <arm_clause> <traj_clause> dest_clause <opt_clauses>
      <sync_clause>
```

If a statement needs more than a single line, commas can be used to end a line after the destination clause or after each optional clause. The reserved word ENDMOVE must then be used to indicate the end of the statement.

Examples appear in the sections that follow.

4.1.1 ARM Clause

Multiple arms can be controlled by a single PDL2 program. Arms are set up at the system level by associating an arm number with a group of axes.

The optional ARM clause designates which arm is to be moved as part of a MOVE statement. For programs that control only a single arm, no designation is necessary.

If specified, the optional arm clause is used as follows:

```
MOVE ARM[1] TO perch
```

The designated arm is used for the entire MOVE statement. Any temporary values assigned in a WITH clause of the move are also applied to the designated arm.

If an arm clause is not included, the default arm is used. The programmer can designate a default arm as a program attribute in the PROGRAM statement, as follows:

```
PROGRAM armtest PROG_ARM=1
.
.
BEGIN
  MOVE TO perch                                -- moves arm 1
  MOVE ARM[2] TO normal                      -- moves arm 2
END armtest
```

If an arm clause is not included and a default arm has not been set up for the program, the value of the predefined variable \$DFT_ARM is used.

4.1.2 TRAJECTORY Clause

The trajectory can be specified either associating the predefined constants JOINT, LINEAR, CIRCULAR to the move statement (for example "MOVE LINEAR TO p1") or assigning them to the \$MOVE_TYPE predefined variable.

The optional trajectory clause designates a trajectory for the motion as part of the MOVE statement syntax, as follows:

```
MOVE trajectory TO perch
```

PDL2 provides the following predefined constants to designate *trajectory*:

```
LINEAR
CIRCULAR
JOINT
```

The trajectory, when specified with the MOVE statement, only affects the motion for which it is designated.

If a trajectory clause is not included in the MOVE statement, the value of the predefined variable \$MOVE_TYPE is used.

The programmer can change the value of \$MOVE_TYPE (JOINT by default) by assigning one of the trajectory predefined constants, as follows:

```
$MOVE_TYPE := JOINT                         -- assigns modal value
MOVE TO perch                            -- joint move
MOVE LINEAR TO slot                      -- linear move
MOVE TO perch                            -- joint move
```

The motions performed by the robot arm can move thru several different trajectories to reach the final position. The trajectory of each motion can be specified as either JOINT,

LINEAR or CIRCULAR. A motion that is specified as having a JOINT trajectory will cause all axis of the robot arm to start and stop moving at the same time. A motion with a LINEAR trajectory will move the Tool Center Point of the robot arm in a straight line from the start position to the end position. A motion that has a CIRCULAR trajectory will move the Tool Center Point of the robot arm in an arc. The described below MOVE TO, MOVE NEAR, MOVE AWAY, MOVE RELATIVE, MOVE ABOUT, MOVE BY, and MOVE FOR statements all require a LINEAR or JOINT trajectory type, and cannot be used with the CIRCULAR trajectory. For more information on the trajectories and motion characteristics, refer to the *Motion Programming* manual.

4.1.3 DESTINATION Clause

The destination clause specifies the kind of move and the destination of the move. It takes one of the following forms:

- **MOVE TO**
TO || destination | joint_list || <VIA_clause>
- **MOVE NEAR**
NEAR destination BY distance
- **MOVE AWAY**
AWAY distance
- **MOVE RELATIVE**
RELATIVE vector IN frame
- **MOVE ABOUT**
ABOUT vector BY distance IN frame
- **MOVE BY**
BY relative_joint_list
- **MOVE FOR**
FOR distance TO destination

Information are also supplied about [VIA Clause](#)

4.1.3.1 MOVE TO

MOVE TO moves the designated arm to a specified destination.

The destination can be any expression resulting in one of the following types:

POSITION
JOINTPOS
XTNDPOS

For example:

```
MOVE LINEAR TO POS(x, y, z, e1, e2, e3, config)
MOVE TO perch
MOVE TO home
```

The destination can also be a joint list. A joint list is a list of real expressions, with each item corresponding to the joint angle of the arm being moved. For example:

```
MOVE TO {alpha, beta, gamma, delta, omega}
-- where alpha corresponds to joint 1, beta to joint 2, etc.
```

Only the joints for which items are listed are moved. For example:

```
MOVE TO { , , gamma, delta}
```

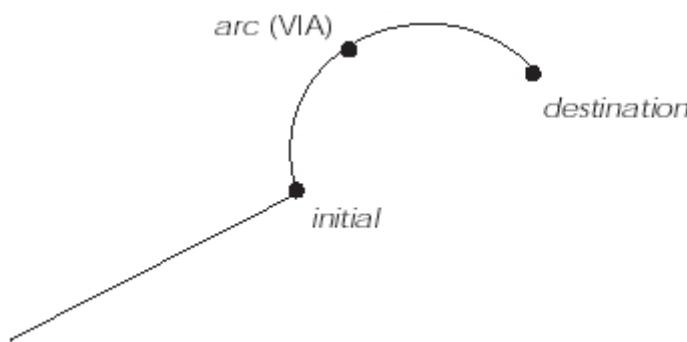
-- only joints 3 and 4 are moved

4.1.3.2 VIA Clause

The optional VIA clause can be used with the MOVE TO destination clause to specify a position through which the arm passes between the initial position and the destination. The VIA clause is used most commonly to define an arc for circular moves as shown in Fig. 4.1 - VIA Position for Circular Move. For example:

```
MOVE TO initial
MOVE CIRCULAR TO destination VIA arc
```

Fig. 4.1 - VIA Position for Circular Move



4.1.3.3 MOVE NEAR

MOVE NEAR allows the programmer to specify a destination along the tool approach vector that is within a specified distance from a position. The distance, specified as a real expression, is measured in millimeters along the negative tool approach vector. The destination can be any expression resulting in one of the following types:

```
POSITION
JOINTPOS
XTNDPOS
```

For example:

```
MOVE NEAR destination BY 250.0
```

4.1.3.4 MOVE AWAY

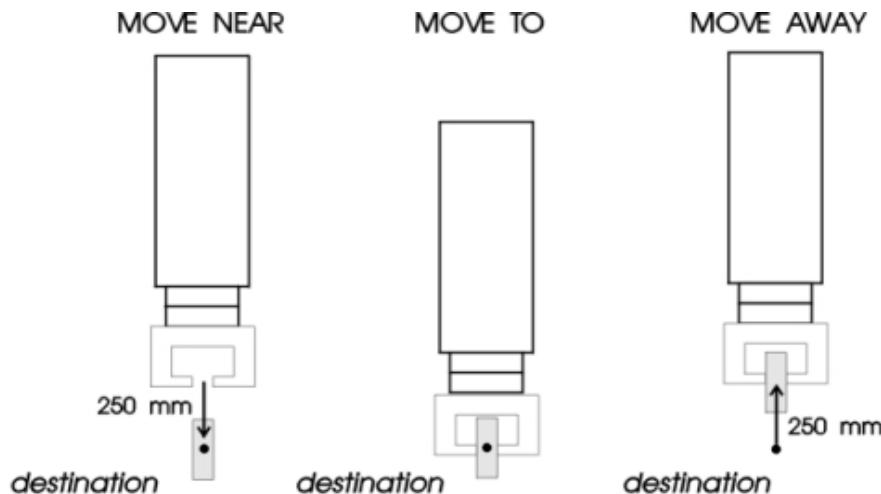
MOVE AWAY allows the programmer to specify a destination along the tool approach vector that is a specified distance away from the current position. The distance, specified as a real expression, is measured in millimeters along the negative tool approach vector.

For example:

```
MOVE AWAY 250.0
```

Fig. 4.2 - MOVE NEAR, TO, and AWAY shows an example of moving near, to, and away from a position.

Fig. 4.2 - MOVE NEAR, TO, and AWAY



4.1.3.5 MOVE RELATIVE

MOVE RELATIVE allows the programmer to specify a destination relative to the current location of the arm. The destination is indicated by a vector expression, measured in millimeters, using the specified coordinate frame.



Note that if one or more optional clauses (arm_clause, trajectory_clause, etc.) are not specified, the corresponding default value is used.

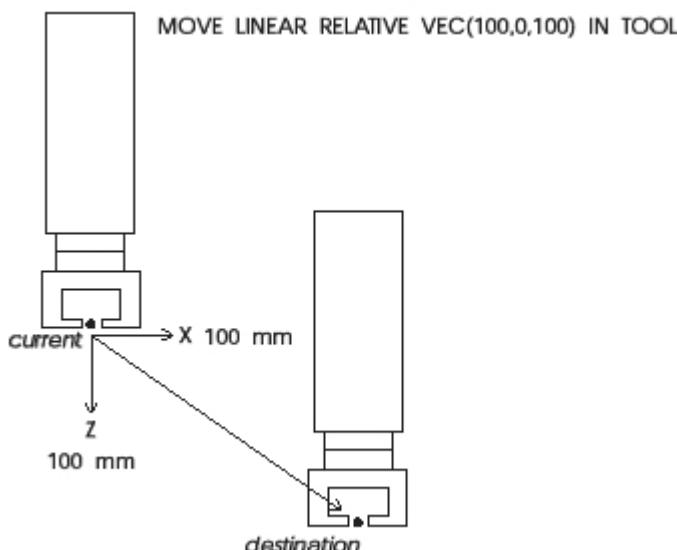
For example:

```
MOVE RELATIVE VEC(100, 0, 100) IN TOOL
MOVE LINEAR RELATIVE VEC(100, 0, 100) IN TOOL
```

The item following the reserved word IN is a frame specification which must be one of the predefined constants TOOL, BASE, or UFRAME.

[Fig. 4.3 - MOVE RELATIVE](#) shows an example of MOVE RELATIVE.

Fig. 4.3 - MOVE RELATIVE



4.1.3.6 MOVE ABOUT

MOVE ABOUT allows the programmer to specify a destination that is reached by rotating the tool an angular distance about a specified vector from the current position. The angle, a real expression, represents the rotation in degrees about the vector, using the specified coordinate frame.

For example:

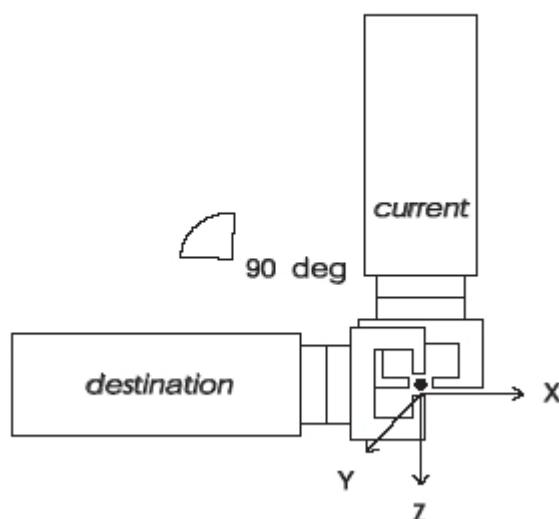
```
MOVE ABOUT VEC(0, 100, 0) BY 90 IN TOOL
```

The item following the reserved word IN is a frame specification which must be one of the predefined constants TOOL, BASE, or UFRAME.

[Fig. 4.4 - MOVE ABOUT](#) shows an example of MOVE ABOUT.

Fig. 4.4 - MOVE ABOUT

```
MOVE ABOUT VEC(0,100,0) BY 90 IN TOOL
```



4.1.3.7 MOVE BY

MOVE BY allows the programmer to specify a destination as a list of REAL expressions, with each item corresponding to an incremental move for the joint of an arm.

For rotational axes, the units are degrees, and for transitional, they are millimeters.

For example:

```
MOVE BY {alpha, beta, gamma, delta, omega}
-- where alpha corresponds to joint 1, beta to joint 2, etc.
```

Only the joints for which items are listed are moved. For example:

```
MOVE BY { , , gamma, , delta}
-- only joints 3 and 5 are moved
```

4.1.3.8 MOVE FOR

MOVE FOR allows the programmer to specify a partial move along the trajectory toward a theoretical destination. The orientation of the tool changes in proportion to the distance.

For example:

MOVE FOR *distance* TO *destination*

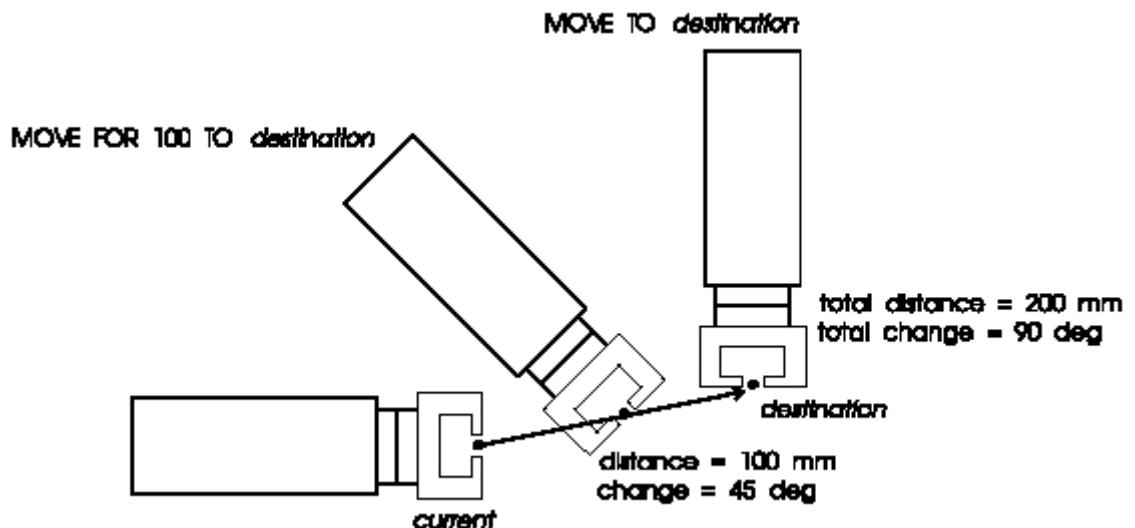
The *distance* is represented by a real expression. If the value is positive, the move is measured toward the destination. If the value is negative, the move is measured in the opposite direction. The distance is measured in millimeters.

The *destination* can be any expression resulting in one of the following types:

POSITION
JOINTPOS
XTNDPOS

Fig. 4.5 - MOVE FOR and MOVE TO shows an example of a MOVE FOR followed by a MOVE TO.

Fig. 4.5 - MOVE FOR and MOVE TO



4.1.4 Optional Clauses

Optional clauses can be used to provide more detailed instructions for the motion. They include the following:

- ADVANCE Clause
- TIL Clause
- WITH Clause

4.1.4.1 ADVANCE Clause

The optional ADVANCE clause takes advantage of the fact that the motion interpolator and the program interpreter run in parallel. The interpreter continues to run the program, even if current motion is still in progress.

When ADVANCE is specified, the interpreter continues program execution as soon as the motion starts. When ADVANCE is not specified, the interpreter waits for the motion to be completed before continuing execution. Program execution can continue up to the next programmed motion for the same arm.

For example:

```

MOVE TO pnt0001p
MyRoutine1()      -- executed after move is completed
MOVE TO pnt0002p ADVANCE
MyRoutine2()      -- executed while move is in progress
MOVE TO pnt0003p
  
```

4.1.4.2 TIL Clause

The optional TIL clause can specify a list of conditions that will cause the motion to be canceled. (Refer to [CANCEL MOTION Statements](#) for more information on canceled motions, and [Chap. Condition Handlers](#) for a description of conditions).

For example:

```
MOVE TO slot TIL $DIN[1] +
```

If the digital signal \$DIN[1] changes to a positive signal during the move, the motion will be canceled.

Conditions are monitored only when the arm is actually in motion. Therefore, the contents of the condition expression are restricted. The conditions cannot be combined using the AND operator and only the following conditions are permitted in a TIL clause (refer to [Chap. Condition Handlers](#) for a complete description of these conditions):

```

AT VIA
TIME n AFTER START -- n is a time in milliseconds
TIME n BEFORE END
DISTANCE n AFTER START -- in cartesian movement only
DISTANCE n BEFORE END -- n is a distance in millimeters
DISTANCE n AFTER VIA
DISTANCE n BEFORE VIA
PERCENT n AFTER START -- in joint movement only
PERCENT n BEFORE END -- n is a number expressing a percentage
-- digital port events digital port states
  
```

4.1.4.3 WITH Clause

The optional WITH clause can designate temporary values for predefined motion variables or enable condition handlers for the duration of the motion.

The WITH clause only affects the motion caused by the current MOVE statement. Previous motions or those that follow are not affected.

The syntax of the WITH clause is as follows:

```
WITH designation <, designation>...
```

where the *designation* is one of the following:

```
|| motion_variable = value | CONDITION[n] ||
```

The following predefined motion variables can be used in a WITH clause of a MOVE statement (refer to [Chap. Predefined Variables List](#) for their meanings):

\$ARM_ACC_OVR	\$ARM_DEC_OVR	\$ARM_LINKED
\$ARM_SENSITIVITY	\$ARM_SPD_OVR	\$AUX_OFST
\$BASE	\$CNFG_CARE	\$COLL_SOFT_PER

\$COLL_TYPE	\$FLY_DIST	\$FLY_PER
\$FLY_TRAJ	\$FLY_TYPE	\$JNT_MTURN
\$JNT_OVR	\$LIN_SPD	\$MOVE_TYPE
\$ORNT_TYPE	\$PAR	\$PROG_ACC_OVR
\$PROG_DEC_OVR	\$PROG_SPD_OVR	\$ROT_SPD
\$SENSOR_ENBL	\$SENSOR_TIME	\$SENSOR_TYPE
\$SFRAME	\$SING_CARE	\$SPD_OPT
\$STRESS_PER	\$TERM_TYPE	TOL_COARSE
\$TOL_FINE	\$TOOL	\$TOOL_CNTR
\$TOOL_FRICTION	\$TOOL_INERTIA	\$TOOL_MASS
\$TOOL_RMT	\$TURN_CARE	\$UFRAME
\$WEAVE_NUM	\$WEAVE_TYPE	\$WV_AMP_PER

Any condition handler that has been defined can be included in a WITH clause. The condition handler is enabled when the motion starts or restarts and disabled when the motion is suspended, canceled, or completed. For further information on condition handlers, refer to [Chap. Condition Handlers](#).

For example:

```
MOVE TO p1 WITH $PROG_SPD_OVR = 50
MOVE TO p1 WITH CONDITION[1]
MOVE TO p1 WITH $PROG_SPD_OVR = 50, CONDITION[1]
```

If a statement needs more than a single line, commas can be used to end a line after a WITH designation. Each new line containing a WITH clause begins with the reserved word WITH and the reserved word ENDMOVE must be used to indicate the end of the statement.

For example:

```
MOVE TO p1 WITH $PROG_SPD_OVR = 50, $MOVE_TYPE = LINEAR,
      WITH CONDITION[1], CONDITION[2], CONDITION[3],
      WITH $TOOL = drive_tool,
ENDMOVE
```

4.1.5 SYNCMOVE Clause

PDL2 allows two arms to be moved simultaneously using the SYNCMOVE clause. This is called a time synchronized move since the arms start and stop together.

For example:

```
MOVE ARM[1] TO part SYNCMOVE ARM[2] TO front
```

The SYNCMOVE clause cannot be used with a [Motion along a PATH](#) statement.

The optional WITH clause can be included as part of the SYNCMOVE clause. The condition handlers included in the WITH clauses will apply to the arm specified in the MOVE or SYNCMOVE clause. For example:

```
MOVE ARM[1] TO part,
      TIL $DIN[1]+,
      TIL DISTANCE 100 BEFORE END,          -- applies to both arms
      WITH CONDITION[1]                   -- applies to arm 1
```

```

SYNCMOVE ARM[2] TO front,
    WITH CONDITION[2],                                -- applies to arm 2
ENDMOVE

```

When the ADVANCE clause is used, it must be placed in the MOVE section and not the SYNCMOVE section.

If an arm is not specified in the SYNCMOVE clause, \$SYNC_ARM is used.



Refer to the Motion Programming manual (chapter 6 - Synchronous Motion (optional feature)) for further information on synchronized motion.

4.1.6 Continuous motion (MOVEFLY)

MOVEFLY and SYNCMOVEFLY can be used in place of the reserved words MOVE and SYNCMOVE to specify continuous motion between Movements of the same type. If another motion follows the MOVEFLY or SYNCMOVEFLY, the arm will not stop at the first destination. The arm will move from the start point of the first motion to the end point of the second motion without stopping on the point that is common to the two motions. For FLY to work properly, the ADVANCE clause must be used to permit interpretation of the following MOVE statement as soon as the first motion begins.



It is not necessary for the two trajectories of the fly motion to have the same Base or Frame, but it is necessary to have the same Tool!

For example:

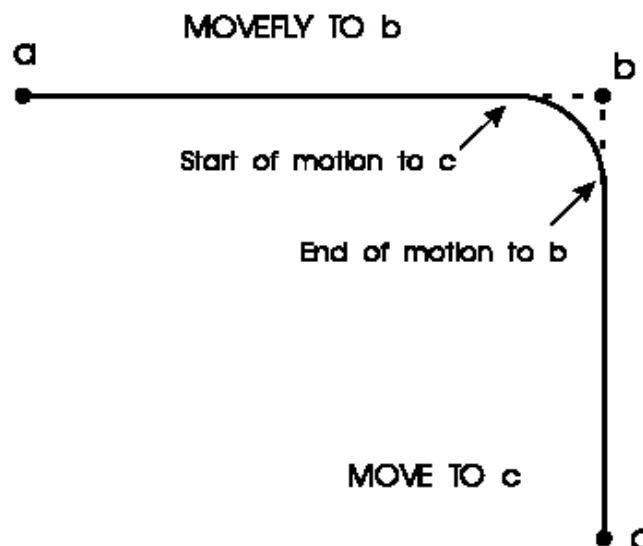
```

MOVE TO a
MOVEFLY TO b ADVANCE
MOVE TO c

```

[Fig. 4.6 - MOVEFLY Between two Cartesian Motions](#) shows the MOVEFLY example.

Fig. 4.6 - MOVEFLY Between two Cartesian Motions



The predefined variable \$FLY_TYPE is used to control the speed of the arm during the

fly motion. If the predefined variable \$FLY_TYPE is set to FLY_NORM (normal fly), the speed of the arm will vary during fly. The FLY_CART modality provides a method of achieving constant speed with an optimal trajectory between two cartesian motions. This option is explained in detail in the *Motion Programming* manual.

The predefined variable, \$FLY_PER, can be used to reduce the time in fly and to bring the trajectory closer to the taught position. The predefined variable \$FLY_PER only affects the arm speed if the predefined variable \$FLY_TYPE is set to FLY_NORM. When \$FLY_PER is in effect, the fly motion will begin at the start of normal deceleration for the motion plus a time equal to 100% minus the percentage specified in \$FLY_PER. For example, if the value of \$FLY_PER is 100%, the fly begins at the start of deceleration of the fly motion. If \$FLY_PER is 75%, then fly will begin after 25% of the declaration is finished (75% will be combined with the next motion.) For more information refer to the *Motion Programming* manual.

When normal non-fly motions are used (MOVE), the stopping characteristics of the motion are determined by the predefined variable, \$TERM_TYPE.

The FLY option must be specified using MOVEFLY. It cannot be specified in the SYNCMOVE section. The program editor will replace SYNCMOVEFLY with SYNCMOVE in the event of a mismatch.

4.1.7 Timing and Synchronization considerations

If the time required for the MOVEFLY motion is shorter than the time required by the interpreter to set up the next MOVE motion, the FLY will not take place. This happens because the motion environment does not get the information it needs in time to perform the FLY.

This situation can occur if the FLY motion is small, meaning the time to move the arm from the current position to the indicated destination is extremely short.

The FLY will also have no affect if additional statements exist between the MOVEFLY statement and the next MOVE statement causing the interpreter to take longer in setting up the next motion.

For correct synchronization between the MOVE statements and other statements, use the optional WITH clause to activate condition handlers. Two examples follow.

To set an output when the fly finishes, use a WITH clause on the MOVEFLY statement to activate a condition handler that sets the output at the end of a fly motion.

```

PROGRAM flycond1
VAR p1, p2, p3 : POSITION
BEGIN
    $DOUT[17] := FALSE
    CONDITION[1] :
        WHEN AT END DO
            $DOUT[17] := TRUE
    ENDCONDITION
    p1 := POS(400, -400, 1900, -93, 78, -62, '')
    p2 := POS(400, 400, 1900, -92, 79, -64, '')
    p3 := POS(800, 400, 1900, -92, 79, -64, '')
    MOVE LINEAR TO p1
    MOVEFLY LINEAR TO p2 ADVANCE WITH CONDITION[1]
    MOVE LINEAR TO p3
END flycond1

```

The output will not be set at *p2* because the MOVEFLY does not reach this position.

Instead, the output will be set where the fly movement ends.

To set an output when the fly starts, use a WITH clause on the next MOVE statement to activate a condition handler that sets the output at the start of a fly motion. For example:

```
PROGRAM flycond2
VAR p1, p2, p3 : POSITION
BEGIN
  $DOUT[17] := FALSE
  CONDITION[1] :
    WHEN AT START DO
      $DOUT[17] := TRUE
  ENDCONDITION
  p1 := POS(400, -400, 1900, -93, 78, -62, '')
  p2 := POS(400, 400, 1900, -92, 79, -64, '')
  p3 := POS(800, 400, 1900, -92, 79, -64, '')
  MOVE LINEAR TO p1
  MOVEFLY LINEAR TO p2 ADVANCE
  MOVE LINEAR TO p3 WITH CONDITION[1]
END flycond2
```

The output will not be set at *p2*, but at the beginning of the fly motion.

4.1.7.1 FLY_CART motion control

FLY_CART (Controller Aided Resolved Trajectory) improves the performance of the controller during cartesian (linear or circular) motions. The speed at the TCP is maintained constant during the fly as long as the machine dynamic solicitations permit. The generated motion will only be affected during a fly between two cartesian motions. Motion is not affected during joint motions. For a detailed discussion of Fly Cart, refer to the *Motion Programming* manual.

4.2 Motion along a PATH

The MOVE ALONG statement specifies movement to the nodes of a PATH variable. The syntax of the MOVE ALONG statement is as follows:

```
MOVE <ARM[n]> ALONG path_var< [ node_range ]> <opt_clauses>
```

If a statement needs more than a single line, commas can be used to end a line after the path specification clause or after each optional clause. The reserved word ENDMOVE must then be used to indicate the end of the statement. Examples appear in the sections that follow.

The MOVE ALONG statement initiates a single motion composed of individual motion segments. A PATH contains a varying number of nodes each of which defines a single motion segment. The MOVE ALONG statement processes each node (or a range of nodes) and moves the arm to the node destination using additional segment information also contained in the node. PATH motion causes the motion environment to generate continuous motion with minimal delay time between the nodes. The beginning of the MOVE ALONG motion is at the start of the first node to be processed and the end of the MOVE ALONG motion is at the termination of the last node to be processed. This is important to understand since the blending of motion caused by successive MOVE statements applies to the beginning and end of the motion. Therefore, the predefined variables such as \$FLY_TYPE and \$TERM_TYPE apply to the last node being processed by the MOVE ALONG statement. PDL2 provides additional predefined

motion variables for handling such information at each segment of a PATH motion. The path node definition can include a set of predefined node fields corresponding to the predefined motion variables which apply to each segment of a PATH motion. The structure of a PATH node is determined by the user-defined node definition contained in the program. The following is a list of these predefined motion variables:

\$CNFG_CARE	\$COND_MASK	\$COND_MASK_BACK
\$JNT_MTURN	\$LIN_SPD	\$MAIN_JNTP
\$MAIN_POS	\$MAIN_XTND	\$MOVE_TYPE
\$ORNT_TYPE	\$ROT_SPD	\$SEG_DATA
\$SEG_FLY	\$SEG_FLY_DIST	\$SEG_FLY_PER
\$SEG_FLY_TRAJ	\$SEG_FLY_TYPE	\$SEG_OVR
\$SEG_REF_IDX	\$SEG_STRESS_PER	\$SEG_TERM_TYPE
\$SEG_TOL	\$SEG_TOOL_IDX	\$SEG_WAIT
\$SING_CARE	\$SPD_OPT	\$TURN_CARE
\$WEAVE_NUM	\$WEAVE_TYPE	

If the node definition does not include a predefined node field, the value specified in a WITH clause is used at each node. If a value is also not specified in a WITH clause, the current value of the corresponding predefined motion variable is used. For example:

```

PROGRAM pth_motn
TYPE lapm_pth = NODEDEF
    $MAIN_POS
    $SEG_OVR
    .
    .
ENDNODEDEF
VAR pth1 : PATH OF lapm_pth
BEGIN
    .
    .
    $TERM_TYPE := COARSE
    $MOVE_TYPE := LINEAR
    MOVE ALONG pth1 WITH $SEG_TERM_TYPE = FINE
    .
    .
END pth_motn

```

In the above example, the segment override (\$SEG_OVR) used in each path segment will be obtained from each path node since this field is included in the node definition. The termination type used in each path segment will be FINE due to the WITH clause. However, the termination type of the last node segment will be COARSE since the value of \$TERM_TYPE applies to the last path node. The motion type (\$MOVE_TYPE) will be LINEAR for the entire path since \$MOVE_TYPE is not a field in the node definition and it is not specified in the WITH clause.

The \$MAIN_POS, \$MAIN_JNTP, and \$MAIN_XTND fields indicate the main destination of a path segment. A node definition can include only one of the \$MAIN_ predefined fields. The particular one chosen indicates the data type of the main destination. In order to use a path in the MOVE ALONG statement, the node definition must include a main destination predefined field.

The \$CNFG_CARE, \$LIN_SPD, \$MOVE_TYPE, \$ORNT_TYPE, \$ROT_SPD, \$SING_CARE, \$SPD_OPT, \$TURN_CARE, and \$WEAVE_NUM fields have the same meaning as the predefined motion variables with the same name. Including them in a

node definition permits these motion parameters to be changed for each path segment.

The \$SEG_TERM_TYPE, \$SEG_FLY_TYPE, and \$SEG_FLY_PER fields have meanings corresponding to the non-segment predefined motion variables. The difference is that they apply to each path segment. The corresponding non-segment predefined motion variables apply to the last node only. If these fields are not included in the node definition and the WITH clause, the corresponding non-segment predefined motion variable is used.

For example:

```

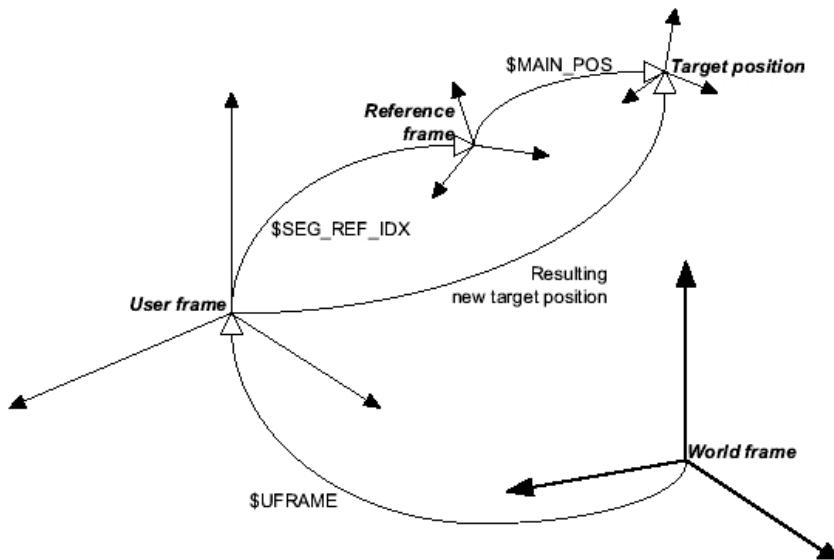
PROGRAM pth_motn
TYPE lapm_pth = NODEDEF
  $MAIN_POS
  $SEG_OVR
  .
  .
  ENDNODEDEF
VAR pth1 : PATH OF lapm_pth
BEGIN
  .
  .
  $TERM_TYPE := FINE
  MOVE ALONG pth1
  .
  .
END pth_motn

```

In the above shown example, since \$SEG_TERM_TYPE is not a field of *lapm_pth* and it is not included in a WITH clause, the value of \$TERM_TYPE (FINE) is used at each segment of the path motion.

The \$SEG_FLY field has the same meaning as the FLY option on the MOVE keyword. It is a boolean and if the value is TRUE, motion to the next node will FLY. This field does not apply to the last node since the FLY option on the MOVE ALONG statement is used. The \$SEG_REF_IDX and \$SEG_TOOL_IDX fields are integers representing indices into the FRM_TBL field of the path. \$SEG_REF_IDX is the index of a frame that can be used to apply an offset to the target position (\$MAIN_POS). This frame is added to the target of the node before being executed. If \$UFRAME is set, it is also added to the result as it happens in every move statement see [Fig. 4.7 - Effects of \\$SEG_REF_IDX definition](#).

Fig. 4.7 - Effects of \$SEG_REF_IDX definition



If \$SEG_REF_IDX is not included in the node definition or if the value of \$SEG_REF_IDX is zero, no reference frame is applied to the path segment. \$SEG_TOOL_IDX is the index of the frame to be used as the tool frame of the path segment. If \$SEG_TOOL_IDX is not included in the node definition or if the value of \$SEG_TOOL_IDX is zero, the value of \$TOOL is used.

A PATH variable contains an array field of 32 INTEGER values called COND_TBL. This table is used for specifying which condition handler will be used during the path motion. For associating a certain condition to a PATH node, the predefined fields \$COND_MASK or \$COND_MASK_BACK must be present in the PATH node definition. \$COND_MASK is used for forward motion to the node and \$COND_MASK_BACK is used for backward motion in the node. These INTEGER fields are bit masks where the individual bits of the \$COND_MASK and \$COND_MASK_BACK value correspond to indices into the COND_TBL array. If a bit is turned on, the condition handler indicated by the corresponding element in the COND_TBL will be enabled for the path segment and automatically disabled when the path segment terminates.

For example:

```

PROGRAM pth
TYPE nd = NODEDEF -- PATH node definition
    $MAIN_POS
    $MOVE_TYPE
    $COND_MASK
    $COND_MASK_BACK
    i : INTEGER
    b : BOOLEAN
ENDNODEDEF
-- The nodes of this path should either be taught or NODE_APPended
VAR p : PATH OF nd
BEGIN
CONDITION[10] :
    WHEN TIME 10 AFTER START DO
        .....
ENDCONDITION
CONDITION[30] :

```

```

WHEN TIME 20 BEFORE END DO
  .....
ENDCONDITION
CONDITION[20] :
  WHEN AT START DO
    .....
ENDCONDITION
.....
p.COND_TBL[1] := 10 -- Initialization of COND_TBL
p.COND_TBL[2] := 15
p.COND_TBL[3] := 20
p.NODE[1].$COND_MASK := 5
p.NODE[4].$COND_MASK_BACK := 2
CYCLE
.....
MOVE ALONG p          -- move forward
MOVE ALONG p[5..1]    -- move backward
.....
END pth

```

In the example above, node 1 has the \$COND_MASK set to 5. Bit 1 and 3 form the value of 5. Therefore, for node 1, the conditions specified in element 1 and 3 of COND_TBL (condition 10 and 20) will be used when moving forward from node 1 to node 2. Node 4 has the \$COND_MASK_BACK set to 2. This means that the condition 15 specified in COND_TBL element 2 will be used when moving backward from node 5 to node 4. Note that the condition handler that are enabled are those corresponding to the program executing the MOVE ALONG statement and not the program containing the MOVE ALONG statement.

Refer also to [Chap. Condition Handlers](#).

The \$SEG_WAIT field is a boolean indicating whether or not processing of the path should be suspended until the path is signalled. This field is used to obtain synchronization between path segment processing and other aspects of the application such as sensor detection, mutual exclusion, etc. If the value of the \$SEG_WAIT field is FALSE, path processing is not suspended at that node. If the value is TRUE, motion for that node will complete but processing of following nodes will be suspended until the path is signalled.

The \$SEG_OVR field indicates the acceleration, deceleration, and speed override for the path segment in a similar way as the \$PROG_ACC_OVR, \$PROG_DEC_OVR and \$PROG_SPD_OVR variables. If \$SEG_OVR is not included in the node definition and the WITH clause of the MOVE ALONG statement, the values of \$PROG_ACC_OVR, \$PROG_DEC_OVR, and \$PROG_SPD_OVR are used for each segment of the path motion.

The \$SEG_TOL field indicates the tolerance for the path segment. This has the same meaning as \$TOL_FINE or \$TOL_COARSE depending on whether the termination type of the path segment is FINE or COARSE. This field does not apply to the last node since the \$TOL_FINE or \$TOL_COARSE will be used based on the value of \$TERM_TYPE which is also applied to the last node.

The \$SEG_DATA field indicates whether the data of the node should be used for the last node segment. If the value is TRUE, the values of the \$SEG_TERM_TYPE, \$SEG_FLY_TYPE, \$SEG_FLY_PER, \$SEG_FLY, and \$SEG_TOL fields are used for the last node segment instead of the values of the corresponding predefined variables.

4.2.1 ARM Clause

The optional arm clause designates which arm is to be moved in the path motion. The arm applies to the main destination field of the nodes. For programs that control only a single arm, no designation is necessary. The optional arm clause is used as follows:

```
MOVE ARM [1] ALONG pth
```

The designated arm is used for the entire MOVE ALONG statement. Any temporary values assigned in a WITH clause of the statement are also applied to the designated arm.

If an arm clause is not included in the MOVE ALONG statement, the default arm is used for the main destination. The programmer can designate a default arm as a program attribute in the PROGRAM statement, as follows:

```
PROGRAM armtest PROG_ARM=1
.
.
.
BEGIN
    MOVE ALONG pth                      --moves arm 1
    MOVE ARM [2] ALONG pth              -- moves arm 2
END armtest
```

If an arm clause is not included and a default arm has not been set up for the program, the value of the predefined variable \$DFT_ARM is used.

4.2.2 NODE RANGE Clause

The optional node range clause allows a path to be started at a node other than node 1. If [node_range] is not present, motion proceeds to the first node of the path, then to each successive node until the end of the path is reached. If [node_range] is present the arm can be moved along a range of nodes specified within the brackets.

The range can be in the following forms:

[n..m]	Motion proceeds to node n of the path, then to each successive node until node m of the path is reached. Backwards motion is allowed by specifying node n greater than node m.
[n..]	Motion proceeds to node n of the path, then to each successive node until the end of the path is reached.

This format allows a path to be executed in any order desired. For example,

```
MOVE ALONG pth[3..10]  -- moves along pth from node 3 to 10
MOVE ALONG pth[5..]    -- moves along pth starting at node 5
MOVE ALONG pth[15..8]  -- moves backwards along pth from node 15 to 8
```

4.2.3 Optional Clauses

Optional clauses can be used to provide more detailed instructions for the path motion. They include the following:

- ADVANCE Clause

- [WITH Clause](#)

4.2.3.1 ADVANCE Clause

The optional ADVANCE clause takes advantage of the fact that the motion interpolator and the program interpreter run in parallel. The interpreter continues to run the program, even if current motion is still in progress.

When ADVANCE is specified, the interpreter continues program execution as soon as the motion starts. When ADVANCE is not specified, the interpreter waits for the motion to be completed before continuing execution. For path motion this means execution will not continue until the last node has been processed. When ADVANCE is specified, program execution can continue up to the next programmed motion for the same arm or the next suspendable statement.

For example:

```
MOVE ALONG pth_1
OPEN HAND 1      -- executed after pth_1 motion is completed

MOVE ALONG pth_1 ADVANCE
OPEN HAND 1      -- executed while pth_1 motion is in progress
MOVE ALONG pth_2 -- executed after pth_1 motion is completed
```

4.2.3.2 WITH Clause

The optional WITH clause can designate temporary values for predefined motion variables for the duration of the motion. The WITH clause affects only the motion caused by the MOVE ALONG statement. Previous motions or those that follow are not affected.

The syntax of the WITH clause is as follows:

WITH designation <, designation>...

where *designation* is:

motion_variable = value

The following predefined motion variables can be used in a WITH clause of a MOVE ALONG statement (refer to the [Chap. Predefined Variables List](#) for their meanings):

\$ARM_ACC_OVR	\$ARM_DEC_OVR	\$ARM_SPD_OVR
\$AUX_OFST	\$BASE	\$CNFG_CARE
\$COLL_TYPE	\$COND_MASK	\$COND_MASK_BACK
\$COND_MASK	\$COND_MASK_BACK	\$FLY_DIST
\$FLY_PER	\$FLY_TRAJ	\$FLY_TYPE
\$JNT_MTURN	\$JNT_OVR	\$LIN_SPD
\$MOVE_TYPE	\$ORNT_TYPE	\$ROT_SPD
\$SEG_DATA	\$SEG_FLY	\$SEG_FLY_DIST
\$SEG_FLY_PER	\$SEG_FLY_TRAJ	\$SEG_FLY_TYPE
\$SEG_OVR	\$SEG_REF_IDX	\$SEG_STRESS_PER
\$SEG_TERM_TYPE	\$SEG_TOL	\$SEG_TOOL_IDX
\$SEG_WAIT	\$SING_CARE	\$SPD_OPT
\$STRESS_PER	\$TERM_TYPE	\$TOL_COARSE

```
$TOL_FINE          $TOOL           $TOOL_CNTR
$TOOL_MASS        $TURN_CARE      $WEAVE_NUM
$WEAVE_TYPE
```

The WITH clause affects only the motion caused by the MOVE ALONG statement. The results of setting the segment related predefined variables will be seen at each path segment only if the corresponding predefined node field is not included in the path node definition. For example:

```
MOVE ALONG pth WITH $SEG_TERM_TYPE = FINE
```

only effects the termination type for each path segment if \$SEG_TERM_TYPE is NOT a field of *pth* nodes.

The non-segment related motion variables included in the WITH clause of a MOVE ALONG statement will only be seen at the last path segment. For example:

```
MOVE ALONG pth WITH $TERM_TYPE = FINE
```

only affects the termination type for the move to the last node in the path. To change any of the motion parameters within a path, the appropriate predefined node field should be used.

If a statement needs more than a single line, commas can be used to end a line after a WITH designation. Each new line containing a WITH clause begins with the reserved word WITH and the reserved word ENDMOVE must be used to indicate the end of the statement.

For example:

```
MOVE ALONG pth WITH $SEG_OVR = 50,
$MOVE_TYPE = LINEAR,
ENDMOVE
```

4.2.4 Continuous Motion (MOVEFLY)

MOVEFLY ALONG can be used in place of the reserved words MOVE ALONG to specify continuous motion. The arm will move from the last node of a path to a point belonging to another motion without stopping on the point that is common to the two motions. If another motion follows the fly, the arm will not stop at the last node processed. The ADVANCE clause should be used to continue execution of the following motion as soon as the first motion begins. For example:

```
MOVEFLY ALONG pth_1 ADVANCE
MOVE TO perch
```

The portion of the first motion that is not covered before the fly begins is determined by the predefined variable \$FLY_PER. The predefined variable \$FLY_TYPE determines whether the motion during the fly will be at a constant speed (FLY_CART) or not (FLY_NORM).

The motion variable \$TERM_TYPE determines the stopping characteristics of the arm for non-continuous motions.

4.3 Stopping and Restarting motions

In addition to the MOVE and MOVE ALONG statements, PDL2 includes statements for stopping and restarting motions. These statements affect the motion state, not the

program state.

4.3.1 CANCEL MOTION Statements

CANCEL CURRENT statement cancels the current motion. A canceled motion cannot be resumed. If any other motions are pending, as a result of the ADVANCE clause or multiple programs, for example, they are executed immediately. The CANCEL ALL statement cancels all motion (current and pending). If a motion being canceled is a path motion, the processing of the path is terminated which means all path segments are canceled.

CANCEL CURRENT SEGMENT or CANCEL ALL SEGMENT statements cancel path segments. A canceled path segment cannot be resumed. If additional nodes remain in the path when the CANCEL CURRENT SEGMENT statement is executed, they are processed immediately. If there are no remaining nodes in the path or the CANCEL ALL SEGMENT statement was used, pending motions (if any) are executed immediately.

CANCEL CURRENT and CANCEL CURRENT SEGMENT cause the arm to decelerate smoothly until the motion ceases. Optionally, the programmer can specify the current motion/segment is to be canceled for the default arm, a list of arms, or for all arms.

```
CANCEL CURRENT
CANCEL CURRENT SEGMENT
CANCEL CURRENT FOR ARM[1], ARM[2]
CANCEL CURRENT SEGMENT FOR ARM[3]
CANCEL CURRENT FOR ALL
CANCEL CURRENT SEGMENT FOR ALL
```

CANCEL ALL and CANCEL ALL SEGMENT cause both the current motion/segment and any pending motions/segments to be canceled. Optionally, the programmer can specify that all motion/segment is to be canceled for the default arm, a list of arms, or for all arms.

```
CANCEL ALL
CANCEL ALL SEGMENT
CANCEL ALL FOR ARM[1], ARM[2]
CANCEL ALL SEGMENT FOR ARM[3]
CANCEL ALL FOR ALL
CANCEL ALL SEGMENT FOR ALL
```

4.3.2 LOCK, UNLOCK, and RESUME Statements

The LOCK statement suspends motion for the default arm, a list of arms, or all arms. When LOCK is executed, the arm smoothly decelerates until the motion ceases.

For example:

```
LOCK
LOCK ARM[1], ARM[2]
LOCK ALL
```

Unlike CANCEL, LOCK prevents pending motions or new motions from starting on the locked arm. The motion can be resumed only by issuing an UNLOCK statement from the same program that issued the LOCK followed by a RESUME statement. The RESUME can be issued from any program. Please note that there shouldn't be any program which attached the arm!

For example:

```

ROUTINE isr      -- interrupt service routine
BEGIN
    .
    .
    UNLOCK
    RESUME
END isr
    .
    .
    CONDITION[1] :
        WHEN $DIN[1]+ DO
            LOCK
            isr
        ENDCONDIDION
    .
    .
    MOVE TO slot WITH CONDITION[1]
    MOVE TO perch

```

The programmer also can specify a list of arms or all arms for the UNLOCK and RESUME statements.

CANCEL motion statements can be used between LOCK and UNLOCK statements to modify the current situation of the issued motions.

4.3.3 SIGNAL SEGMENT Statement

The SIGNAL SEGMENT statement resumes path motion that is currently suspended. Path motion will be suspended if the \$SEG_WAIT field of a node is TRUE. The only way to resume the path motion is to execute a SIGNAL SEGMENT statement.

For example:

```
SIGNAL SEGMENT pth
```

The \$SEG_WAIT field of a path node is a boolean indicating whether or not processing of the path should be suspended until the path is signalled. This field is used to obtain synchronization between path segment processing and other aspects of the application such as sensor detection, mutual exclusion, etc. If the value of the \$SEG_WAIT field is FALSE, path processing is not suspended at that node.

If the SIGNAL SEGMENT statement is executed and the path specified is not currently suspended, the statement will have no effect. However, a trappable error will occur. Unlike semaphores, SEGMENT signals are not counted which means extra SIGNALs on paths are not remembered. Therefore, the SIGNAL SEGMENT statement must be executed after the path is suspended by \$SEG_WAIT.

4.3.4 HOLD Statement

The HOLD statement places running holdable programs in the ready state and causes motion to decelerate to a stop. The HOLD statement works exactly like the HOLD button on the TP and operation panel.

4.4 ATTACH and DETACH Statements

The ATTACH and DETACH statements are used to control program access to a device. This is useful when multiple executing programs require the use of the same device. Robot arms are examples of such devices.

When applied to an arm device, the ATTACH statement requests exclusive motion

control of an arm. If the arm is already attached to a program or it is currently being used in a motion, an error will occur. The ATTACH statement can be used to attach the default arm, a list of arms, or all arms.

```
ATTACH ARM
ATTACH ARM[1], ARM[2]
ATTACH ARM ALL
```

Once a program has attached an arm, only that program can initiate motion for the attached arm. If a MOVE or RESUME statement is issued from a different program, it is an error causing the program to be paused.

In addition to the ATTACH statement, a program can attach an arm using the ATTACH attribute on the PROGRAM statement. In this case, the PROG_ARM is attached when the program is activated. If that arm is currently attached to another program or it is currently being used in a motion, the program will not be activated. If the programmer doesn't want the PROG_ARM to be attached when the program is activated, the DETACH attribute must be specified on the PROGRAM statement. The default is to attach the PROG_ARM. (Refer to [Statements List](#) chapter for further details on the PROGRAM statement and its attributes.)

The DETACH statement terminates exclusive motion control of the default arm, a list of arms, or all arms currently attached by the program.

```
DETACH ARM
DETACH ARM[1], ARM[2]
DETACH ARM ALL
```

CANCEL motion statements can be issued by any program while the arm is attached.

4.5 HAND Statements

End-of-arm tooling such as hands can be controlled with HAND statements. Two hands are available per arm, corresponding to the TP T1 and T2 keys. The programmer designates, using the HAND configuration tool (SH_INST) delivered with the system software, which hand is to be operated as part of the HAND statement.

PDL2 includes the following HAND statements:

```
OPEN HAND hand_num
CLOSE HAND hand_num
RELAX HAND hand_num
```

For example:

```
OPEN HAND 1
CLOSE HAND 2
RELAX HAND 2
```

The programmer can also designate a particular arm, a list of arms, or all arms as follows:

```
OPEN HAND 1 FOR ARM[n]
CLOSE HAND 2 FOR ARM[1], ARM[2]
RELAX HAND 1 FOR ALL
```



Note that, if the HAND has not been configured yet, the pressure of T1 and T2 will not set any output by default.

The following types of hands can be controlled:

Single Line

- OPEN sets the output line to the active value;
- CLOSE sets the output line to the inactive value;
- RELAX is the same as CLOSE.

Dual Line

- OPEN sets line 1 to the active value and line 2 to the inactive value;
- CLOSE sets line 2 to the active value and line 1 to the inactive value;
- RELAX sets lines 1 and 2 to the inactive value.

Pulse

- OPEN sets line 1 to the active value, waits a delay time, and sets line 1 to the inactive value;
- CLOSE sets line 2 to the active value, waits a delay time, and sets line 2 to the inactive value;
- RELAX is the same as CLOSE.

Step

- OPEN sets line 1 to the active value, waits a delay time, and sets line 2 to the active value;
- CLOSE sets line 2 to the inactive value, waits a delay time, and sets line 1 to the inactive value;
- RELAX is the same as CLOSE.

5. PORTS

5.1 General

Configuration and access to I/O points, is based on a Device-centric architecture (i.e. this Device has these I/O points) as opposed to an I/O-centric architecture (i.e. this Input point is mapped to this Device).

The configuration is based on a hierarchical approach, as opposed to a “fixed” array.

The System I/O configuration information, is described by a tree structure, assigned to [\\$CIO_TREEID: Tree identifier for the configure I/O tree](#) predefined variable.

The configuration tree is used by the System at the powerup, to initialize all predefined variables related to Devices and I/Os ([\\$IO_DEV: Input/Output Device Table](#), [\\$NUM_IO_DEV: Number of Input/Output Devices](#), [\\$IO_STS: Input/Output point StatusTable](#), [\\$NUM_IO_STS: Number of Input/Output points](#)).

Current chapter explains the existing types of ports used in PDL2, associated to the I/O points of the Devices connected to the Controller Unit.

The available Port types are as follows:

- digital, flexible, and analog Ports, configured by the user to accommodate specific application I/Os ([par. 5.2 User-defined and Appl-defined Ports on page 104](#));
- system-defined Ports, internally mapped for system Devices such as operator devices, arms, and timers ([par. 5.3 System-defined Ports on page 105](#));
- other Ports used by PDL2 programs to communicate one another ([par. 5.4 Other Ports on page 118](#)).

The following [Tab. 5.1 - Ports table](#) lists the I/O-related Ports used by PDL2.

Tab. 5.1 - Ports table

Identifier	Data Type	Description	Category
\$DIN	BOOLEAN	digital input	user-def
\$DOUT	BOOLEAN	digital output	user-def
\$IN	BOOLEAN	digital input	appl-def
\$OUT	BOOLEAN	digital output	appl-def
\$GI	BOOLEAN	digital input	sys-def
\$GO	BOOLEAN	digital output	sys-def
\$AIN	INTEGER	analog input	user-def
\$AOUT	INTEGER	analog output	user-def
\$SDI	BOOLEAN	digital input	sys-def
\$SDI32	INTEGER	digital input	sys-def
\$FMI	INTEGER	flexible multiple input	appl-def
\$FMO	INTEGER	flexible multiple output	appl-def
\$FMI_BIT	BOOLEAN	digital input	appl-def
\$FMO_BIT	BOOLEAN	digital output	appl-def

Tab. 5.1 - Ports table (Continued)

Identifier	Data Type	Description	Category
\$SDO	BOOLEAN	digital output	sys-def
\$SDO32	INTEGER	digital output	sys-def
\$FDIN	BOOLEAN	digital input	sys-def
\$FDOUT	BOOLEAN	digital output	sys-def
\$HDIN	BOOLEAN	digital input	sys-def
\$TIMER (ms)	INTEGER	system timers	sys-def
\$TIMER_S (s)	INTEGER	system timers	sys-def
\$BIT	BOOLEAN	shared memory	PDL2-PLC
\$WORD	INTEGER	shared memory	PDL2-PLC
\$WORD_BIT	BOOLEAN	shared memory	PDL2-PLC
\$PROG_TIMER_xx	INTEGER	program timers	sys-def

Depending on the type of array, each item in a Port array represents a particular input or output signal, or a shared memory location. For user-defined Ports, the signal that corresponds to a particular item depends on how the I/O is configured.

Also refer to [Predefined Variables List](#) chapter for further details about these variables.

As with any array, individual items are **accessed using an index**, as shown in the following examples:

```

FOR n := top TO bottom DO
    $DOUT[n] := OFF
ENDFOR

REPEAT
    check_routine
UNTIL $AIN[34] > max

body_type := $FMI[3]
SELECT body_type OF
CASE(1):
    $FMO[1] := four_door
CASE(2):
    $FMO[2] := two_door
CASE(3):
    $FMO[3] := hatch_back
ELSE:
    type_error
ENDSELECT

IF $FDIN[21] = ON THEN -- U1 button on programming terminal
    $FDOUT[21] := ON -- U1 LED on programming terminal
ENDIF

```

A program can always obtain the value of a Port. The value received from reading an Output Port corresponds to the last value that was written to that Port.

A program can assign values to user-defined Output Ports and the system-defined

\$FDOUT Ports.

5.2 User-defined and Appl-defined Ports

Digital, flexible and analog Ports are configured by the user to accommodate specific application I/Os.



NOTE THAT the user is allowed to access the help string definition of a port via PDL2 program, by means of the [DV_CNTRL Built-In Procedure](#), code 14, which returns an index to access the related \$IS_HELP_STR field within [\\$IO_STS: Input/Output point StatusTable structure](#).

Example:

```
s := 'DOUT_5'
DV_CNTRL(14, (s), i)
WRITE LUN_CRT ($IO_STS[i].IS_HELP_STR, NL)
```

A detailed description follows about:

- [\\$DIN and \\$DOUT](#)
- [\\$IN and \\$OUT](#)
- [\\$AIN and \\$AOUT](#)
- [\\$FMI and \\$FMO](#)

5.2.1 \$DIN and \$DOUT

\$DIN and \$DOUT Ports allow a program to access data on a single (discrete) digital input or output signal. PDL2 treats this data as a BOOLEAN value.

For example:

```
IF $DIN[n] = ON THEN
  .
  .
  .
$DOUT[n] := ON
```

5.2.2 \$IN and \$OUT

\$IN and \$OUT are digital Ports, reserved to application programs.

5.2.3 \$AIN and \$AOUT

\$AIN and \$AOUT Ports allow a program to access data in the form of an analog signal. PDL2 treats this data as an INTEGER value.

For example:

```
IF $AIN[n] > 8 THEN
  .
  .
  .
ENDIF
$AOUT[n] := 4
```

The system converts the analog input signal to a 16-bit binary number and a 16-bit binary number to an analog output signal.

5.2.4 \$FMI and \$FMO

\$FMI and \$FMO are Flexible Multiple Ports, reserved to application programs. PDL2

treats them as INTEGER values.

These Ports can also be read as BITS: in such a situation, they are named

- \$FMI_BIT, \$FMO_BIT.

They are always referred to as groups of 32 bits, not depending on the corresponding \$FMx dimension.

The bits of the \$FMx Ports are to be referred to in the following way:

- for \$FMx[1] - \$FMx_BIT[1]..\$FMx_BIT[32],
- for \$FMx[2] - \$FMx_BIT[33]..\$FMx_BIT[64],
- etc.

Note that \$FMx Ports are analog Ports, whereas \$FMx_BIT Ports are digital ones.

5.3 System-defined Ports

System-defined Ports are internally mapped for system devices such as operator devices, arms, and timers.

A detailed description follows about:

- [\\$SDI](#) and [\\$SDO](#)
- [\\$SDI32](#) and [\\$SDO32](#)
- [\\$GI](#) and [\\$GO](#)
- [\\$FDIN](#) and [\\$FDOUT](#)
- [\\$HDIN](#)
- [\\$TIMER](#), [\\$TIMER_S](#)
- [\\$PROG_TIMER_xx](#).

5.3.1 \$SDI and \$SDO

\$SDI and \$SDO Ports allow a program read-only access to data on system-defined signals as if they were single input or output lines. PDL2 treats this data as a BOOLEAN value.

Here follows a list of \$SDI signal meanings:

\$SDI	Meaning
1	Ext Emergency Stop latched
2	General Stop latched
3	Enabling Device latched
4	Auto Stop latched
5	TP Wireless STOP latched
6	TP Wireless Ena Dev latched
7	Data valid latched
8	Mode Selector latched
9	Recoverable diagnostics
10	Not recoverab. diagnostics
11..16	reserved
17	Ext Emergency Stop checked

18	General Stop checked
19	Enabling Device checked
20	Auto Stop checked
21	Wireless Stop checked
22	Wireless Ena. Dev. checked
23	Wireless telegram is valid
24	mode selector is valid
25	Recoverab. diagn. status
26..31	reserved
32	Delayed emerg. in progress
33	Countdown timer1
34	Countdown timer2
35	Countdown timer3
36	Countdown timer4
37	Programming mode 1
38	Programming mode 2
39	Automatic mode Local
40	Automatic mode Remote
41	Ext EmergencyStop channel 1
42	Ext EmergencyStop channel 2
43	General stop channel 1
44	General stop channel 2
45	Enabling device channel 1
46	Enabling device channel 2
47	Auto stop channel 1
48	Auto stop channel 2
49	Wireless stop channel 1
50	Wireless stop channel 2
51	Wireless Ena. dev. chann. 1
52	Wireless Ena. dev. chann. 2
53	Brake Release Dev. NC
54	Optional pwr contactors NC
55	reserved
56	EnT2 output status
57	EnT1 output status
58	V24Safe output status
59	Ext Emerg.Repeater status
60	Auto Stop Repeater status
61	Ena. Dev. Repeater status
62	AUTO mode Repeater status
63	PROG mode Repeater status
64	Wireless Stop Repeater stat
65	Forced Unpair switch

66	Pairing Switch
67	WiTP on Docking Station
68	Telegram answer delay 1
69	Telegram answer delay 2
70	Telegram answer delay 3
71	Pairing permission from TP
72	Unpairing permission from TP
73	TP Wired configuration
74	TP Wireless configuration
75	Wireless exclusion active
76	TP Wireless paired
77	TP Wired connected
78	Pairing state 1
79	Pairing state 2
80	Pairing state 3
81	Test of inputs OK
82	Wired/Wireless Config OK
83	Time Switch match OK
84	V24SAFE Status OK
85	EnT1 Status OK
86	KEN_NC and BRD_NC Status OK
87	Forced unpairing done
88	Forced unpairing rejected
89	EnT2 Status OK
90	Output test OK
91	ERM1 Status OK
92	ERM2 Status OK
93	POWER ON self test OK
94	Wireless Stop Repeater OK
95	CPU mismatch OK
96	Watchdog OK
97	Mode selector inputs OK
98	Enabling Device inputs OK
99	X30 inputs OK
100	LStMon input OK
101..104	reserved
105	Docking Station link OK
106	FIA5 PowerControl OK
107	24V OK on X100 supply
108	24V OK after JP200
109	24V OK on V24 SAFE
110	24V OK on V24 DRIVE-OFF
111	24V OK on V24 I/O

112	24V OK on V24 INT
113	overload on V24 I/O or SAFE
114	overload on V24 INT
115-116	reserved
117	UPS ready
118	UPS buffering
119	UPS replace battery
120	Local Stop monitor
121	User Input 1
122	User Input 2
123	User Input 3
124	User Input 4
125	User Input 5
126	User Input 6
127	User Input 7
128	User Input 8

Here follows a list of \$SDO signal meanings.

\$SDO	Meaning
1	DRIVE ON
2	C5G Ready for Pairing
3	C5G Ready for Unpairing
4	disable 24V of APS
5	disable UPS Battery
6	disable EnT2 motor breaking
7-8	reserved
9	cabinet Fan speed 1
10	cabinet Fan speed 2
11	cabinet Fan speed 3
12	cabinet Fan speed 4
13..16	reserved
17	User Output 1
18	User Output 2
19	User Output 3
20	User Output 4
21..24	reserved
25	V24 INT Forced to OFF
26	V24 DRIVE OFF Forced to OFF
27	V24 I/O Forced to OFF
28-30	reserved
31	V24 WiTP on DS Forced to OFF

32	V24 AP on DS Forced to OFF
33	Led1 on DS green color 1
34	Led1 on DS green color 2
35	Led1 on DS red color 1
36	Led1 on DS red color 2
37..40	reserved
41	searching of Wireless Device completed
42	connecting device found
43	connecting device status OK
44	C5G power up completed
45	pairing/unpairing error code bit 1
46	pairing/unpairing error code bit 2
47	pairing/unpairing error code bit 3
48	reserved
49	Serial word to WiTP bit 1
50	Serial word to WiTP bit 2
51	Serial word to WiTP bit 3
52	Serial word to WiTP bit 4
53	Serial word to WiTP bit 5
54	Serial word to WiTP bit 6
55	Serial word to WiTP bit 7
56	Serial word to WiTP bit 8
57	Serial word to WiTP bit 9
58	Serial word to WiTP bit 10
59	Serial word to WiTP bit 11
60	Serial word to WiTP bit 12
61	Serial word to WiTP bit 13
62	Serial word to WiTP bit 14
63	Serial word to WiTP bit 15
64	Serial word to WiTP bit 16

5.3.2 \$SDI32 and \$SDO32

These Ports allow to group 32 sequential \$SDI/\$SDO BOOLEAN elements in just one INTEGER element.

Example:

\$SDI32[1] includes \$SDI[1] .. \$SDI[32],
\$SDI32[2] includes \$SDI[33] .. \$SDI[64], etc.

5.3.3 \$GI and \$GO

\$GI and \$GO Ports allow a program to refer to general signals as if they were single input or output lines. PDL2 treats this data as a BOOLEAN value.

A table summarizing the \$GI and \$GO available in the System, is shown below:

\$GI	Meaning
1..8	reserved
9	IESK ARM 1 enabled
10	IESK ARM 2 enabled
11	IESK ARM 3 enabled
12	IESK ARM 4 enabled
13..16	reserved
17	DRIVE ON
18	DRIVE OFF
19	START
20	HOLD
21	U1
22	U2
23	U3
24	U4
25	CANCEL ALARM
26	Safety speed
27..32	reserved
33	IESK 1st AUX ARM 1 enabled
34	IESK 2nd AUX ARM 1 enabled
35	IESK 3rd AUX ARM 1 enabled
36	IESK 4th AUX ARM 1 enabled
37	IESK 1st AUX ARM 2 enabled
38	IESK 2nd AUX ARM 2 enabled
39	IESK 3rd AUX ARM 2 enabled
40	IESK 4th AUX ARM 2 enabled
41	IESK 1st AUX ARM 3 enabled
42	IESK 2nd AUX ARM 3 enabled
43	IESK 3rd AUX ARM 3 enabled
44	IESK 4th AUX ARM 3 enabled
45	IESK 1st AUX ARM 4 enabled
46	IESK 2nd AUX ARM 4 enabled
47	IESK 3rd AUX ARM 4 enabled
48	IESK 4th AUX ARM 4 enabled
49	IEAK ARM 1 MOTOR ON
50	IEAK ARM 2 MOTOR ON
51	IEAK ARM 3 MOTOR ON
52	IEAK ARM 4 MOTOR ON
53	IEAK ARM 1 MOTOR OFF
54	IEAK ARM 2 MOTOR OFF
55	IEAK ARM 3 MOTOR OFF

56	IEAK ARM 4 MOTOR OFF
57	IEAK ARM 1 closed contacts
58	IEAK ARM 2 closed contacts
59	IEAK ARM 3 closed contacts
60	IEAK ARM 4 closed contacts

\$GO	Meaning
1	DRIVE ON internal command
2	WiTP pairing available
4	alarm
5	START from remote panel
6	Reset from remote panel
7	local emergency STOP
8	forced I/O
9	IESK ARM 1 enable
10	IESK ARM 2 enable
11	IESK ARM 3 enable
12	IESK ARM 4 enable
13..16	reserved
17	alarm
18	DRIVE ON status
19	START/HOLD on mot prog
20	REMOTE
21	teach enabled(DRI.ON+T1)
22	U1
23	U2
24	U3
25	U4
26	NO active latch alarm
27	Safety speed (for Remote)enabled
28	programming mode(PROGR)
29	reserved
30	state selector in local or remote
31	system ready
32	Heart Bit
33	system in STANDBY
34	active JOG
35	motion program in execution
36	EME/GEN/AUTO STOP active
37	WinC5G on PC connected
38	kind of TP
39	TP disconnected in PROGR state

40	state selector on T1P
41	recovery from power down
42	TP on cabinet
43	enabling device pressed
44-45	reserved
46	state selector on REMOTE
47	state selector on AUTO
48	state selector on T1
49	HOLD from remote
50	HOLD pressed on TP
51	reserved
52	DRIVE OFF from REMOTE
53	DRIVE OFF pressed on TP
54	reserved
55	TP connected
56	ALARM state due to high severity
57	REMOTE state
58	START key pressed
59	system in ALARM state
60	system in programming state
61	system in automatic
62	system in HOLD
63	DRIVEs state
64	reserved
65	reference position 1
66	reference position 2
67	reference position 3
68	reference position 4
69	reference position 5
70	reference position 6
71	reference position 7
72	reference position 8
73..80	reserved
81	Collision alarm
82	CollisionDetect enabled
83	Collision alarm (2)
84	CollisionDetect enabled(2)
85..96	reserved
97	IESK 1st AUX ARM 1 enable
98	IESK 2nd AUX ARM 1 enable
99	IESK 3rd AUX ARM 1 enable
100	IESK 4th AUX ARM 1 enable
101	IESK 1st AUX ARM 2 enable

102	IESK 2nd AUX ARM 2 enable
103	IESK 3rd AUX ARM 2 enable
104	IESK 4th AUX ARM 2 enable
105	IESK 1st AUX ARM 3 enable
106	IESK 2nd AUX ARM 3 enable
107	IESK 3rd AUX ARM 3 enable
108	IESK 4th AUX ARM 3 enable
109	IESK 1st AUX ARM 4 enable
110	IESK 2nd AUX ARM 4 enable
111	IESK 3rd AUX ARM 4 enable
112	IESK 4th AUX ARM 4 enable
113	IEAK ARM 1 MOTOR ON
114	IEAK ARM 2 MOTOR ON
115	IEAK ARM 3 MOTOR ON
116	IEAK ARM 4 MOTOR ON

5.3.4 \$FDIN and \$FDOUT

The \$FDIN and \$FDOUT Ports allow a program to access data on system-defined signals as if they were single input or output lines. These signals correspond to selectors, function keys and LEDs on the Operator Panel, Teach Pendant, and PC keyboard (WinC5G Program is active). PDL2 treats this data as a BOOLEAN value.

The following tables list \$FDIN and \$FDOUT signal meanings.

When the word “Setup” is used in the following explanations, it refers to the Setup of REC key that can also be selected using either the Setup Page on Teach Pendant or the menus of programming environments.

The FDOUTs related to the TP cannot be assigned at the PDL2 program level. Some led's are handled ONLY when working in the programming environments. These are marked with an asterisk in the “Element” column.

Tab. 5.2 - \$FDIN

Element	Associated Input
\$FDIN[1..8]	reserved
\$FDIN[9]	state selector on REMOTE
\$FDIN[10]	state selector on AUTO
\$FDIN[11]	state selector on T1
\$FDIN[12..15]	reserved
\$FDIN[16]	reserved
\$FDIN[17]	DRIVE ON softkey
\$FDIN[18]	DRIVE OFF softkey
\$FDIN[19]	START button
\$FDIN[20]	HOLD button
\$FDIN[21]	U1 softkey
\$FDIN[22]	U2 softkey

Tab. 5.2 - \$FDIN (Continued)

\$FDIN[23]	U3 softkey
\$FDIN[24]	U4 softkey
\$FDIN[25]	reserved
\$FDIN[26]	reserved
\$FDIN[27]	Enabling Device pressed
\$FDIN[28..32]	reserved
\$FDIN[33]	F1 softkey (**)
\$FDIN[34]	F2 softkey (**)
\$FDIN[35]	F3 softkey (**)
\$FDIN[36]	F4 softkey (**)
\$FDIN[37]	F5 softkey (**)
\$FDIN[38]	F6 softkey (**)
\$FDIN[39]	F7 softkey (**)
\$FDIN[40]	F8 softkey (**)
\$FDIN[41]	PREV/TOP (**)
\$FDIN[42]	SCRN (**)
\$FDIN[43]	CHAR (**)
\$FDIN[44]	DEL (**)
\$FDIN[45]	up arrow (**)
\$FDIN[46]	left arrow (**)
\$FDIN[47]	down arrow (**)
\$FDIN[48]	right arrow (**)
\$FDIN[49]	SEL/SCRL (**)
\$FDIN[50]	ENTER (**)
\$FDIN[51]	CUT softkey (**)
\$FDIN[52]	COPY softkey (**)
\$FDIN[53]	/ softkey (**)
\$FDIN[54]	S.NXT softkey (**)
\$FDIN[55]	SRCH softkey (**)
\$FDIN[56]	PASTE softkey (**)
\$FDIN[57]	MODE softkey (**)
\$FDIN[58]	UNDEL softkey (**)
\$FDIN[59]	DEL softkey (**)
\$FDIN[60]	PAGE softkey (**)
\$FDIN[61]	MARK softkey (**)
\$FDIN[62]	HELP (**)
\$FDIN[63..64]	reserved
\$FDIN[65]	COORD button
\$FDIN[66]	reserved
\$FDIN[67]	%+
\$FDIN[68]	%-
\$FDIN[69..72]	reserved

Tab. 5.2 - \$FDIN (Continued)

\$FDIN[73]	+1X
\$FDIN[74]	+2Y
\$FDIN[75]	+3Z
\$FDIN[76]	+4
\$FDIN[77]	+5
\$FDIN[78]	+6
\$FDIN[79]	AUX A+
\$FDIN[80]	AUX B+
\$FDIN[81]	-1X
\$FDIN[82]	-2Y
\$FDIN[83]	-3Z
\$FDIN[84]	-4
\$FDIN[85]	-5
\$FDIN[86]	-6
\$FDIN[87]	AUX A-
\$FDIN[88]	AUX B-
\$FDIN[89]	EXCL (**)
\$FDIN[90]	T1
\$FDIN[91]	T2
\$FDIN[92]	REC
\$FDIN[93]	MOD softkey (**)
\$FDIN[94]	BACK
\$FDIN[95]	RUN softkey (**)
\$FDIN[96]	reserved
\$FDIN[97]	POS softkey (**)
\$FDIN[98]	JNTP softkey (**)
\$FDIN[99]	XTND softkey (**)
\$FDIN[100]	JNT softkey (**)
\$FDIN[101]	LIN softkey (**)
\$FDIN[102]	CIRC softkey (**)
\$FDIN[103]	reserved
\$FDIN[104]	FLY softkey (**)
\$FDIN[105]	reserved
\$FDIN[106]	A1 softkey (**)
\$FDIN[107]	A2 softkey (**)
\$FDIN[108]	RESET softkey (**)
\$FDIN[109..128]	reserved
\$FDIN[129]	F1 on PC keyboard (WinC5G Program active)
\$FDIN[130]	F2 on PC keyboard (WinC5G Program active)
\$FDIN[131]	F3 on PC keyboard (WinC5G Program active)
\$FDIN[132]	F4 on PC keyboard (WinC5G Program active)
\$FDIN[133]	F5 on PC keyboard (WinC5G Program active)

Tab. 5.2 - \$FDIN (Continued)

\$FDIN[134]	F6 on PC keyboard (WinC5G Program active)
\$FDIN[135]	F7 on PC keyboard (WinC5G Program active)
\$FDIN[136]	F8 on PC keyboard (WinC5G Program active)
\$FDIN[137..158]	reserved
\$FDIN[159]	JPAD Up (along Z axis)
\$FDIN[160]	JPAD Down (along Z axis)
\$FDIN[161]	JPAD North (internal, approaching the user - along X axis)
\$FDIN[162]	JPAD South (external, moving away - along X axis)
\$FDIN[163]	JPAD East (right - along Y axis)
\$FDIN[164]	JPAD West (left - along Y axis)

(**) In case of Teach Pendant, these \$FDIN/\$FDOUT detect variations only if TP-INT Page or Virtual Keyboard are active.

Tab. 5.3 - \$FDOUT

Element	Associated Input
\$FDOU[1..16]	reserved
\$FDOU[17]	DRIVE ON led
\$FDOU[18..20]	reserved
\$FDOU[21]	U1
\$FDOU[22]	U2
\$FDOU[23]	U3
\$FDOU[24]	U4
\$FDOU[25]	ALARM led. When an alarm occurs, this led is lighted
\$FDOU[26]	T1 led for HAND 1 of the selected arm. It is lighted when the HAND is closed
\$FDOU[27]	T2 led for HAND 2 of the selected arm. It is lighted when the HAND is closed
\$FDOU[28]*	FLY led. This led gets lighted when the Setup of the MOVE statement is set to MOVEFLY or when the FLY key on TP is pressed (**)
\$FDOU[29]	enables the Save Screen function on the TP display
\$FDOU[30]	reserved
\$FDOU[31]*	POSITION variable led. This led is lighted when the Setup of the Variable is set to POSITION, or when the POS key on the TP is pressed (**)
\$FDOU[32]*	JNTPos variable led. This led gets lighted when the Setup of the Variable is set to JOINTPOS or when the JNTP key on the TP is pressed (**)
\$FDOU[33]*	XTNDpos variable. This led is lighted when the Setup of the Variable is set to XTNDPOS or when the XTND key on the TP is pressed (**)
\$FDOU[34]*	JNT trajectory led. This led is lighted when the Setup of the Trajectory is set to JOINT or when the JNT key on the TP is pressed (**)

Tab. 5.3 - \$FDOUT (Continued)

\$FDOUT[35]*	LIN trajectory led. This led is lighted when the Setup of the Trajectory is set to LINEAR or when the LIN key on TP is pressed (**)
\$FDOUT[36]*	CIRC trajectory led. This led gets lighted when the Setup of the Trajectory is set to Circular or when the CIRC key is pressed (**)

(**) In case of Teach Pendant, these \$FDIN/\$FDOUT detect variations only if T-PINT Page or Virtual Keyboard are active.

5.3.5 \$HDIN

The \$HDIN Port allows a program read-only access to a motion event port. Items in the array correspond to high speed digital input signals.

\$HDIN is a special Port monitored by the motion environment. Any input on one of these lines triggers a hardware interrupt signal. The motion environment reacts according to how the port is set up. For example, it could cause an Arm to stop and/or record its current position.

There are two \$HDIN Ports for each Arm, in the System. Thus, it is possible to simultaneously handle the available inputs for each Arm.

The user is responsible for installing the connection from the \$HDIN Port to detection devices to use with this Port.

The association between Arm and \$HDIN index is as follows:

- \$HDIN[1] channel 1 for ARM 1
- \$HDIN[2] channel 1 for ARM 2
- \$HDIN[3] channel 1 for ARM 3
- \$HDIN[4] channel 1 for ARM 4
- \$HDIN[5] channel 2 for ARM 1
- \$HDIN[6] channel 2 for ARM 2
- \$HDIN[7] channel 2 for ARM 3
- \$HDIN[8] channel 2 for ARM 4

Such Ports are updated at the I/O scanning time (usually 10 ms).

The [HDIN_SET Built-In Procedure](#) and [HDIN_READ Built-In Procedure](#) are used for handing the \$HDIN from a PDL2 program. Also the [\\$HDIN_SUSP: temporarily disables the use of \\$HDIN](#) system variable must be considered.

Refer to the [BUILT-IN Routines List](#) and [Predefined Variables List](#) chapters for further details.

As far as concerns the association between HSI Inputs on the Controller and \$HDIN Ports in PDL2 Programs, please reference to the following [Tab. 5.4](#).

Tab. 5.4 - Association between HSI inputs and \$HDIN Ports

HSI Signal	Related ARM	Associated Port
HSI1	- ARM1	- \$HDIN[1]
	- ARM2	- \$HDIN[2]
	- ARM3	- \$HDIN[3]
	- ARM4	- \$HDIN[4]

Tab. 5.4 - Association between HSI inputs and \$HDIN Ports

HSI Signal	Related ARM	Associated Port
HSI3	- ARM1	- \$HDIN[5]
	- ARM2	- \$HDIN[6]
	- ARM3	- \$HDIN[7]
	- ARM4	- \$HDIN[8]

5.3.6 \$TIMER, \$TIMER_S

The \$TIMER Port allows a program read and write access to a system timer. The array index corresponds to individual timers. Their values are expressed as INTEGER values measured in milliseconds. The system timers run continuously.



The maximum allowed value before \$TIMER and \$TIMER_S overflow is $2^{31} = 2147483648$, which means about 24 days for \$TIMER Ports (milliseconds) and about 68 years for \$TIMER_S Ports (seconds).

\$TIMER is accessible by all running programs. The [ATTACH Statement](#) can be used to disallow read and write access to a timer by other program code. When a timer is attached, only code belonging to the same program containing the [ATTACH Statement](#) can access the timer. The [DETACH Statement](#) is used to release exclusive control of a timer. The value of the timer is not changed when it is attached or detached. For more information, refer to the description of the ATTACH & DETACH statements in [Statements List](#) chapter.

\$TIMER values are preserved during Power Failure Recovery: no \$TIMER value is lost. A timer overflow generates trappable error 40077.

5.3.7 \$PROG_TIMER_XX

These timers are similar to \$TIMER and \$TIMER_S, but they belong to PDL2 Programs.

The following PROG_TIMERS are available:

- \$PROG_TIMER_O and \$PROG_TIMER_OS - they belong to the program owner. The first one is in milliseconds and the second one is in seconds.
- \$PROG_TIMER_X and \$PROG_TIMER_XS - they belong to the executing program (calling program). The first one is in milliseconds and the second one is in seconds.



The maximum allowed value before \$PROG_TIMER_O, \$PROG_TIMER_X, \$PROG_TIMER_OS and \$PROG_TIMER_XS overflow is $2^{31} = 2147483648$, which means about 24 days for \$PROG_TIMER_O and \$PROG_TIMER_X Ports (milliseconds) and about 68 years for \$PROG_TIMER_OS and \$PROG_TIMER_XS Ports (seconds).

5.4 Other Ports

Such Ports are used by PDL2 programs to communicate with one another.

A detailed description follows about:

- \$BIT
- \$WORD

5.4.1 \$BIT

The \$BIT Port allows a program to access to a bit; PDL2 treats this as BOOLEAN data. The maximum index value for \$BIT Ports is 1024.

5.4.2 \$WORD

The \$WORD Port allows a program to access to a word; PDL2 treats this as an INTEGER data, where only the 16th significative bits of each word are meaningful.

These Ports can also be read as BITS: in such a situation they are named

- \$WORD_BIT.

They are referred to as groups of 16 bits.

The bits of the \$WORD Ports are to be referred to in the following way:

- for \$WORD[1] - \$WORD_BIT[1]..\$WORD_BIT[16],
- for \$WORD[2] - \$WORD_BIT[17]..\$WORD_BIT[32],
- etc.

Note that in case of \$WORD_BIT, PDL2 treats them as BOOLEAN data.

5.5 System State After Restart

This section describes the state of the C5G system after it has been restarted. There are two different methods of restarting. They are cold start (on the Teach Pendant: Start Page, Restart softkey or ConfigureControllerRestart command from WinC5G) and power failure recovery. When the Controller undergoes any of these restarting processes, the state that the system comes up in will be different depending on the type of restart. What is meant by the “state of the system”, is the state of the \$DIN, \$IN, \$BIT, etc. The state of the system is described after each type of shown below restarting.

The \$SDI, \$SDO, \$GI, \$GO Ports are updated to the current state of the system. As far as concerns the other I/O Ports (\$DIN, \$IN, \$BIT, \$WORD; etc.), the following sections describe which is the state they assume after restart.

5.5.1 Cold Start

When a cold start is issued from the Controller, using either the Restart softkey in the Start Page or the ConfigureCntrlerRestartCold command (CCRC) from within TP-INT Page, the system will come back up as indicated below.

\$DIN/\$DOUT/\$IN/\$OUT/\$FMI/\$FMO	cleared
\$BIT	cleared
\$AIN/\$AOUT	cleared
\$WORD	cleared
PDL2 program	deactivated and erased from memory

Note that forced \$DIN, \$DOUT, \$IN, \$OUT, \$AIN, \$AOUT, \$FMI, \$FMO are cleared also if they were previously forced.

5.5.2 Power Failure

When a power failure occurs, the system will return as indicated below.

\$DIN/\$DOUT/\$IN/\$OUT/ \$FMI/\$FMO/\$AIN/\$AOUT	cleared. Note that if they are configured as retentive, they are frozen and not cleared.
\$BIT	frozen
\$AIN/\$AOUT	frozen
\$WORD	frozen
PDL2 program	saved (a non-holdable program continues where it left off; for an holdable program, you must also turn the drives ON and press the START key to resume motion)
\$TIMER/\$TIMER_S/ \$PROG_TIMER_xx	frozen

Forced \$DIN, \$DOUT, \$IN, \$OUT, \$AIN, \$AOUT, \$FMI, \$FMO are frozen upon a power failure.

6. EXECUTION CONTROL

Current chapter describes PDL2 statements that control the order in which statements are executed within a program and the order in which multiple programs are executed.

PDL2 language provides:

- Flow Control Statements
- Program Control Statements
- Program Synchronization Statements
- Program Scheduling facilities

6.1 Flow Control

Statements that control the execution flow within a program are called Flow Control Statements.

The following Flow Control Statements are available:

- IF Statement
- SELECT Statement
- FOR Statement
- WHILE Statement
- REPEAT Statement
- GOTO Statement



It is strongly recommended to avoid using cycles in the program which stress too much the CPU. For example, REPEAT..UNTIL cycle with an UNTIL condition which is always FALSE (so that the cycle is always repeated) and WAIT FORs inside which the condition is always true. A simplified representation of such situation is :

```
vb_flag := TRUE
```

```
vb_flag2 := FALSE
```

```
REPEAT
```

```
    WAIT FOR vb_flag -- condition always true
    UNTIL vb_flag2 -- infinite loop
```

Cycles like this can cause a nested activity of the processes in the system software making it difficult to get the CPU by low priority tasks. If this situation lasts for a long period, the error “65002-10 : system error Par 2573” can occur. It is recommended, in this case, to change the logic of the program.

Within a program, execution normally starts with the first statement following the BEGIN statement and proceeds until the END statement is encountered. PDL2 provides statements to alter this sequential execution in the following ways:

- choosing alternatives:

- IF statement,
- SELECT statement;
- looping:
 - FOR statement,
 - WHILE statement,
 - REPEAT statement;
- unconditional branching:
 - GOTO statement.

6.1.1 IF Statement

The IF statement allows a program to choose between two possible courses of action, based on the result of a BOOLEAN expression.

If the expression is TRUE, the statements following the IF clause are executed. Program control is then transferred to the statement following the ENDIF.

If the expression is FALSE, the statements following the IF clause are skipped and program control is transferred to the statement following the ENDIF.

[Fig. 6.1 - IF Statement Execution](#) shows the execution of an IF statement.

An optional ELSE clause, placed between the last statement of the IF clause and the ENDIF, can be used to execute some statements if the BOOLEAN expression is FALSE.

[Fig. 6.2 - ELSE Clause Execution](#) shows the execution of an IF statement with the optional ELSE clause.

The syntax for an IF statement is as follows:

```
IF bool_exp THEN
  <statement>...
<ELSE
  <statement>...>
ENDIF
```

Fig. 6.1 - IF Statement Execution

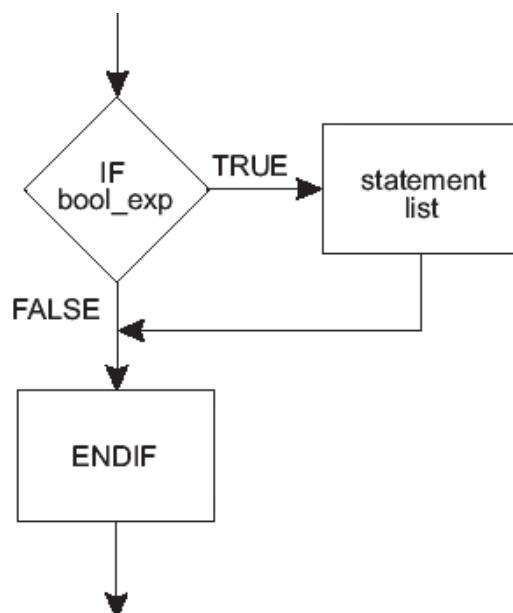
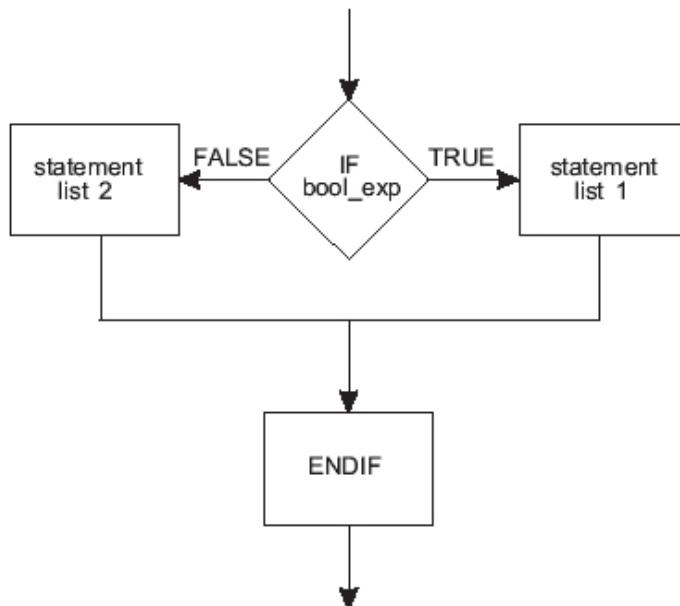


Fig. 6.2 - ELSE Clause Execution



Any executable PDL2 statements can be nested inside of IF or ELSE clauses, including other IF statements.

Examples of the IF statement:

```

IF car_num =3 THEN
    num_wheels :=4
ENDIF

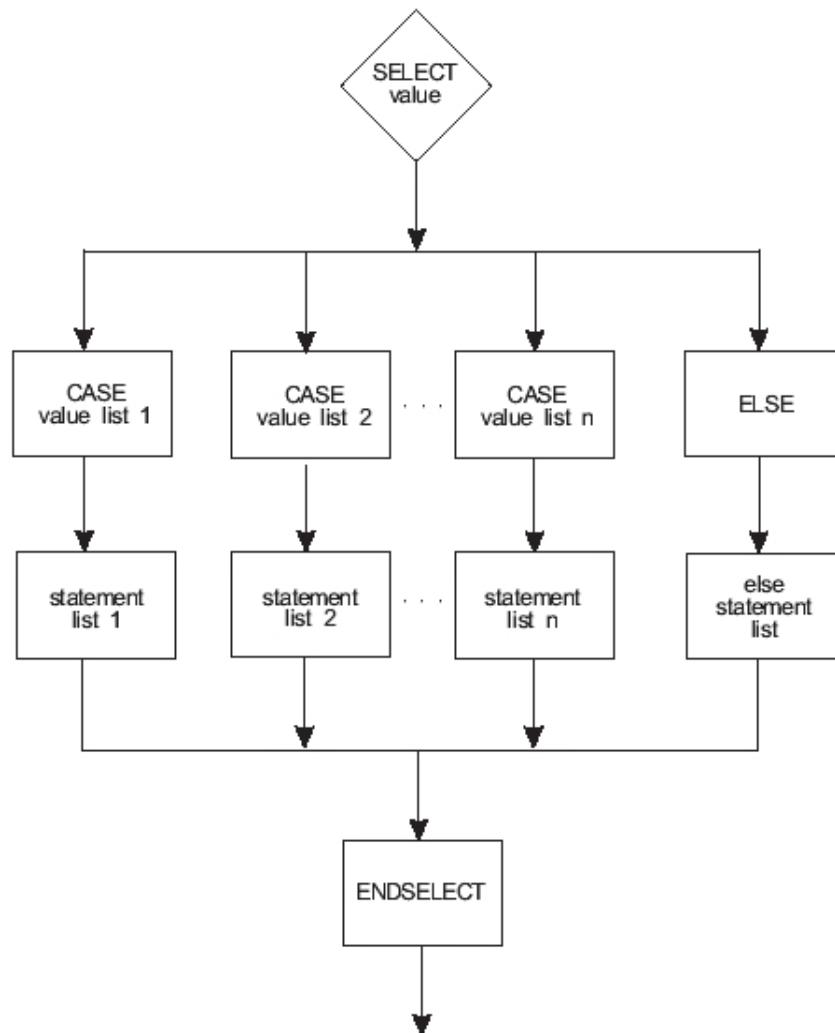
IF part_ok THEN
    good_parts := good_parts + 1
ELSE
    bad_parts := bad_parts + 1
ENDIF
    
```

6.1.2 SELECT Statement

The SELECT statement allows a program to choose between several alternatives, based on the result of an INTEGER expression. Each of these alternatives is referred to as a case.

When the INTEGER expression is evaluated, it is compared to the INTEGER value of each case, looking for a match. When a match is found, the statements corresponding to that case are executed and program control transfers to the statement following the ENDSELECT statement. Any remaining cases are skipped. [Fig. 6.3 - SELECT Statement Execution](#) shows the execution of a SELECT statement.

Fig. 6.3 - SELECT Statement Execution



An optional ELSE clause, placed between the last case and the ENDSELECT statement, can be used to execute a series of statements if the INTEGER expression does not match any of the case values. Without this ELSE clause, a failure to match a case value results in an error.

The syntax for a SELECT statement is as follows:

```

SELECT int_exp OF
  CASE (int_val <, int_val>...):
    <statement>...
  <CASE (int_val <, int_val>...):
    <statement>... >...
  <ELSE:
    <statement>...>
ENDSELECT
  
```

The INTEGER values in each case can be predefined or user-defined constants or literals. Expressions are not allowed. In addition, the same INTEGER value cannot be used in more than one case.

Example of a SELECT statement:

```

SELECT tool_type OF
  
```

```

CASE (1):
    $TOOL := utool_weld
    style_weld
CASE (2):
    $TOOL := utool_grip
    style_grip
CASE (3):
    $TOOL := utool_paint
    style_paint
ELSE:
    tool_error
ENDSELECT

```

6.1.3 FOR Statement

The FOR statement is used when a sequence of statements is to be repeated a known number of times. It is based on an INTEGER variable that is initially assigned a starting value and is then incremented or decremented each time through the loop until an ending value is reached or exceeded. The starting, ending and step values are specified as INTEGER expressions.

[Fig. 6.4 - FOR Statement Execution](#) shows the execution of a FOR statement.



The usage of a GOTO statement inside of a FOR loop is not recommended

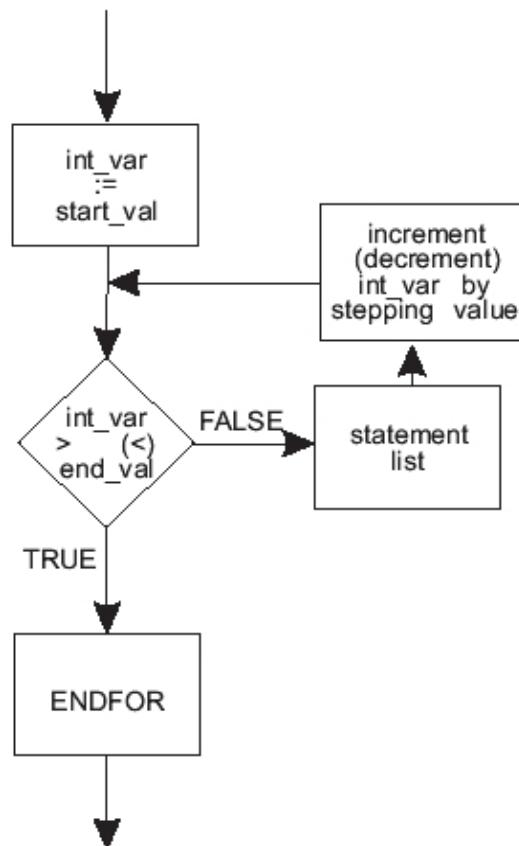
Before each loop iteration, the value of the INTEGER variable is compared to the ending value. If the loop is written to count up, from the starting value TO the ending value, then a test of less than or equal is used. If this test initially fails, the loop is skipped. If the STEP value is one, the loop is executed the absolute value of (the ending value - the starting value + 1) times.

If the loop is written to count down, from the starting value DOWNTO the ending value, a test of greater than or equal is used. If this test initially fails, the loop is skipped. If the STEP value is one, the loop is executed the absolute value of (the ending value - the starting value - 1) times.



Even if the starting and ending values are equal, the loop is still executed one time

An optional STEP clause can be used to change the stepping value from one to a different value. If specified, the INTEGER variable is incremented or decremented by this value instead of by one each time through the loop. The STEP value, when using either the TO or DOWNTO clause, should be a positive INTEGER expression to ensure that the loop is interpreted correctly.

Fig. 6.4 - FOR Statement Execution

The syntax of a FOR statement is as follows:

```

FOR int_var := start_val || TO | DOWNTTO || end_val <STEP step_val>
DO
  <statement>...
ENDFOR
  
```

Example of a FOR statement:

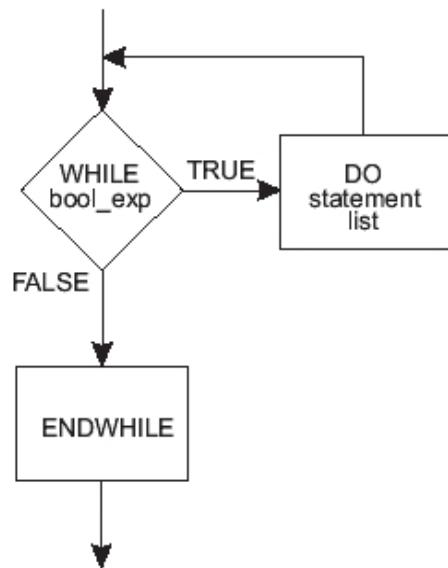
```

FOR test := 1 TO num_hoses DO
  IF pressure[test] < 170 THEN
    WRITE('Low pres. in hose #', test, 'Test for leaks.', NL)
  ENDIF
ENDFOR
  
```

6.1.4 WHILE Statement

The WHILE statement causes a sequence of statements to be executed as long as a BOOLEAN expression is true. If the BOOLEAN expression is FALSE when the loop is initially reached, the sequence of statements is never executed (the loop is skipped). Fig. 6.5 - WHILE Statement Execution shows the execution of a WHILE statement.

Fig. 6.5 - WHILE Statement Execution



The syntax for a WHILE statement is as follows:

```

WHILE bool_exp DO
    <statement>...
ENDWHILE
    
```

Example of a WHILE statement:

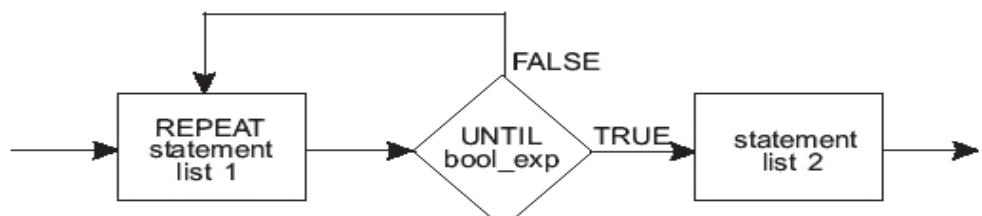
```

WHILE num < num_errors DO
    IF priority_index[num] < 2 THEN
        WRITE (err_text[num], '(non critical)', NL)
    ELSE
        WRITE (err_text[num], ' ***CRITICAL*** ', NL)
    ENDIF
    num := num + 1
ENDWHILE
    
```

6.1.5 REPEAT Statement

The REPEAT statement causes a sequence of statements to be executed until a BOOLEAN expression becomes TRUE. This loop is always executed at least once, even if the BOOLEAN expression is TRUE when the loop is initially encountered. Fig. 6.6 - REPEAT Statement Execution shows the execution of a REPEAT statement.

Fig. 6.6 - REPEAT Statement Execution



The syntax for a REPEAT statement is as follows:

```

REPEAT
    
```

```

<statement>...
UNTIL bool_exp

```

Example of a REPEAT statement:

```

REPEAT
  WRITE ('Exiting program', NL)           -- statement list 1
  WRITE ('Are you sure? (Y/N) : ')
  READ (ans)
UNTIL (ans = Y) OR (ans = N)
IF ans = Y THEN
  DEACTIVATE prog_1                     -- statement list 2
ENDIF

```

6.1.6 GOTO Statement

The GOTO statement causes an unconditional branch. Unconditional branching permits direct transfer of control from one part of the program to another without having to meet any conditions.

In most cases where looping or non-sequential program flow is needed in a program, it can be done with other flow control statements discussed in this chapter.

Unconditional branch statements should be used only when no other control structure will work.

[Fig. 6.7 - GOTO Statement Execution](#) shows the execution of a GOTO statement.

The GOTO statement transfers program control to the place in the program specified by a statement label, a constant identifier at the left margin.

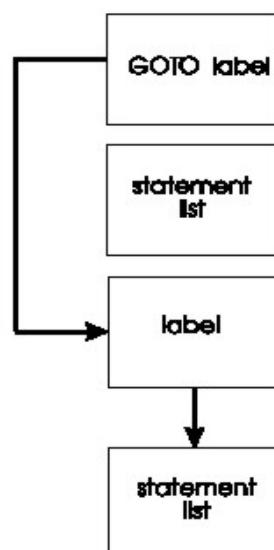
The statement LABEL is followed by two consecutive colons (::). Executable statements may follow on the same line, or any line after the label.

The syntax for a GOTO statement is as follows:

```
GOTO statement_label
```

GOTO statements should be used with caution. The label must be within the same routine or program body as the GOTO statement. Do not use a GOTO statement to jump into or out of a structured statement, especially a FOR loop, because this will cause a run-time error.

[Fig. 6.7 - GOTO Statement Execution](#)



Example of a GOTO statement:

```

PROGRAM prog_1
VAR jump : BOOLEAN
BEGIN
    .
    .
    IF jump THEN
        GOTO here
    ENDIF
    .
    .
    here::: WRITE ('This is where the GOTO transfers to')
    .
END prog_1

```

6.2 Program Control

Statements that control execution of entire programs are called Program Control Statements.

The following Program Control Statements are available:

- ACTIVATE Statement
- PAUSE Statement
- UNPAUSE Statement
- DEACTIVATE Statement
- CYCLE and EXIT CYCLE Statements
- DELAY Statement
- WAIT FOR Statement
- BYPASS Statement

Program control statements alter the state of a program. A program can check and change its own state and the state of other programs. Program control statements can also create a continuous cycle and exit from that cycle.

Programs can be divided into two categories, depending on the holdable/non-holdable program attribute.

- holdable programs are controlled by START and HOLD. Usually, holdable programs include motion, but that is not a requirement;
- non-holdable programs are not controlled by START and HOLD. Generally, they are used as process control programs. Non-holdable programs cannot contain motion statements, however, they can use positional variables for other purposes. The motion statements that are not allowed are RESUME, CANCEL, LOCK, UNLOCK, and MOVE.

The HOLD or NOHOLD attribute can be specified in the PROGRAM statement to indicate a holdable or non-holdable program. The default program attribute is HOLD.

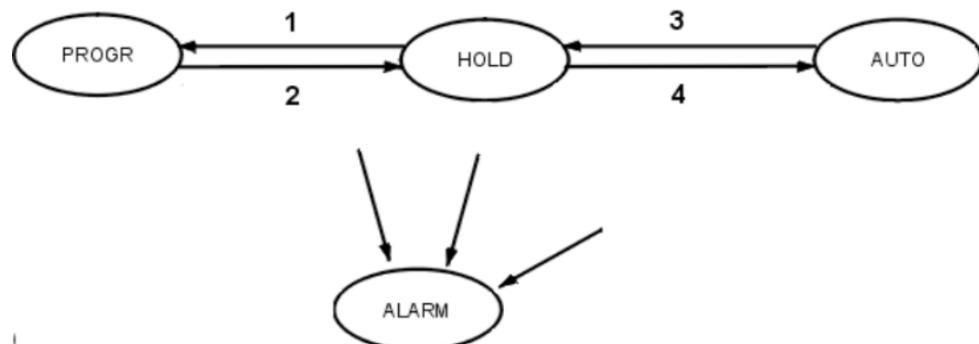
States for activated programs include running, ready, paused, and paused-ready with the following meaning:

- running is the state of a program that is currently executing;
- paused is the state entered by a program when its execution is interrupted by a PROGRAM STATE PAUSE command or by the PAUSE statement;

- ready is the state entered by an holdable program when it is ready to run but needs the pressure of the START button for being executed;
- paused - ready is the state entered by a program when the described before conditions related to the paused and ready states are true. For running the program from this state, it is needed to issue either the PROGRAM STATE UNPAUSE command or the UNPAUSE statement and then press the START button.

[Fig. 6.8 - Program States](#) shows the program states and the actions that cause programs to change from one state to another.

Fig. 6.8 - Program States



1. State selector on REMOTE position
2. State selector on T1 position + HOLD button released
3. HOLD or DRIVES OFF or state selector position changing
4. HOLD or DRIVES OFF or state selector position changing
5. State selector on AUTO or REMOTE + HOLD button released

PROGR = programming
 AUTO = AUTO, REMOTE

6.2.1 PROG_STATE Built-In Function

The PROG_STATE built-in function allows a program to check its own state and the state of other programs. Its calling sequence is as follows:

```
PROG_STATE (prog_name_str)
```

prog_name_str can be any valid string expression. This function returns an INTEGER value indicating the program state of the program specified by *prog_name_str*. For a list of the values returned by this function, refer to the "Built-In Routine List" chapter.

6.2.2 ACTIVATE Statement

Programs that are loaded into memory can be concurrently activated by the ACTIVATE statement. Only one activation of a given program is allowed at a given time. The effect of activating a program depends on the holdable/non-holdable attribute of the program issuing the statement and the program being activated. If the statement is issued from a non-holdable program, holdable programs are placed in the ready state and non-holdable programs are placed in the running state. If the statement is issued from a holdable program, the programs are placed in the running state.

The ACTIVATE statement syntax is as follows:

```
ACTIVATE prog_name <, prog_name>...
```

For example:

```
ACTIVATE weld_main, weld_util, weld_ctrl
```

When a program is activated, the following occurs for that program:

- initialized variables are given their initial values
- if it is a holdable program without the DETACH attribute, the arm is attached and if it is a non-holdable program with the ATTACH attribute the arm is attached.

6.2.3 PAUSE Statement

The PAUSE statement causes a program to be paused until an UNPAUSE operation is executed for that program.

Pausing a program that is running puts that program in a paused state. Pausing a program that is in a ready state puts that program in a paused-ready state. The following events continue even while a program is paused:

- current and pending motions;
- condition handler scanning;
- current output pulses;
- current and pending system calls;
- asynchronous statements (DELAY, PULSE, WAIT, etc.).

The PAUSE statement syntax is as follows:

```
PAUSE < || prog_name <, prog_name>... | ALL || >
```

If a *prog_name* or list of names is specified, those programs are paused. If no name is included, the program issuing the statement is paused. If ALL is specified, all running and ready programs are paused. The statement has no effect on programs that are not executing.

For example:

```
PAUSE
PAUSE weld_main, weld_util, weld_ctrl
PAUSE ALL
```

6.2.4 UNPAUSE Statement

The UNPAUSE statement unpauses paused programs. The effect of unpausing a program depends on the holdable/non-holdable program attribute.

If the statement is issued from a non-holdable program, holdable programs are placed in the ready state and non-holdable programs are placed in the running state. If the statement is issued from a holdable program, the programs are placed in the running state.

The UNPAUSE statement syntax is as follows:

```
UNPAUSE || prog_name <, prog_name>... | ALL ||
```

If a *prog_name* or list of names is specified, those programs are unpause. If ALL is specified, all paused programs are unpause. The statement has no effect on programs

that are not paused.

For example:

```
UNPAUSE weld_main, weld_util, weld_cntrl
UNPAUSE ALL
```

6.2.5 DEACTIVATE Statement

The DEACTIVATE statement deactivates programs that are in running, ready, paused, or paused-ready states. Deactivated programs remain loaded, but do not continue to cycle and cannot be resumed. They can be reactivated with the ACTIVATE statement.

When a program is deactivated, the following occurs for that program:

- current and pending motions are canceled;
- condition handlers are purged;
- the program is removed from any lists (semaphores);
- reads, pulses, and delays are canceled;
- current and pending system calls are aborted;
- opened files are closed;
- attached devices, timers, and condition handlers are detached;
- locked arms are unlocked but the motion still needs to be resumed.

The DEACTIVATE statement syntax is as follows:

```
DEACTIVATE < || prog_name <, prog_name>... | ALL ||
```

If a *prog_name* or list of names is specified, those programs are deactivated. If the program name list is empty, the program issuing the statement is deactivated. If ALL is specified, all executing programs are deactivated.

For example:

```
DEACTIVATE
DEACTIVATE weld_main, weld_util, weld_cntrl
DEACTIVATE ALL
```

6.2.6 CYCLE and EXIT CYCLE Statements

The CYCLE statement can either be a separate statement or an option on the BEGIN statement.

It allows the programmer to create a continuous cycle. When the program END statement is encountered, execution continues back at the CYCLE statement.

This continues until the program is deactivated.

The CYCLE statement is only allowed in the main program. The CYCLE statement cannot be used inside a routine. A program can contain only one CYCLE statement.

The CYCLE statement syntax is as follows:

```
CYCLE
```

The BEGIN statement syntax using the CYCLE option is as follows:

```
BEGIN CYCLE
```

The EXIT CYCLE statement causes program execution to skip the remainder of the

current cycle and immediately begin the next cycle. An exited cycle cannot be resumed. Exiting a cycle cancels all pending and current motion, cancels all outstanding asynchronous statements, and resets the program stack.



NOTE that the EXIT CYCLE statement does NOT reset the current working directory and the \$PROG_ARG predefined variable (which contains the line argument passed in when the program was activated).

Exiting a cycle does NOT close files, detach resources, disable or purge condition handlers, or unlock arms.

Consequently, it is more powerful than a simple GOTO statement.

The EXIT CYCLE statement can be used in the main program as well as routines.

The EXIT CYCLE statement syntax is as follows:

```
EXIT CYCLE < || prog_name <, prog_name... | ALL || >
```

If a *prog_name* or list of names is specified, those programs exit their current cycles. If *prog_name* is not specified, the program issuing the statement exits its current cycle. If ALL is specified, all executing programs exit their current cycles.

6.2.7 DELAY Statement

The DELAY statement causes execution of the program issuing it to be suspended for a specified period of time, expressed in milliseconds.

The following events continue even while a program is delayed:

- current and pending motions;
- condition handler scanning;
- current pulses;
- current and pending system calls.

The DELAY statement syntax is as follows:

```
DELAY int_expr
```

The *int_expr* indicates the time to delay in milliseconds.

6.2.8 WAIT FOR Statement

The WAIT FOR statement causes execution of the program issuing it to be suspended until the specified condition is satisfied. The syntax is as follows:

```
WAIT FOR cond_expr
```

This statement uses the same *cond_expr* allowed in a WHEN clause of a condition handler (as listed in [Condition Handlers](#) chapter).

6.2.9 BYPASS Statement

If a certain program is executing a suspendable statement, BYPASS can be used for skipping that statement and continuing the execution from the next line. A suspendable statement is one of the following: READ, WAIT FOR, WAIT on a SEMAPHORE, SYS_CALL, DELAY, PULSE, MOVE.

The BYPASS statement syntax is:

```
BYPASS < || prog_name <, <prog_name>... | ALL ||><flags>
```

If a *prog_name* or a list of names is specified, the execution of those programs will continue on the line following the suspendable statement that is currently being executed. If ALL is specified, the BYPASS will apply to all executing programs. If no program is specified, the program issuing the statement will bypass itself, this last case has only meaning when the BYPASS is issued from a condition handler action or in an interrupt service routine (refer to “Condition handlers” chapter for further details).

flags is an optional mask that can be used for specifying which suspendable statement should be bypassed. For example, 64 is the value for bypassing the READ statement. Refer to the “Built-in Routine List” chapter for the list of the values to be used for the *flags* field.

For determining if a program is suspended on a certain statement, the PROG_STATE built-in function can be called and then, the integer value returned by this function can be passed to the BYPASS statement in correspondence to the field flags. For example:

```
state:= PROG_STATE (weld_main)
IF state = 64 THEN
  BYPASS weld_main 64
ENDIF
```

6.3 Program Synchronization

Statements that control multiple programs execution are called Program Synchronization Statements.

Multiple programs executing at the same time are synchronized using shared semaphore variables. This provides a method for mutual exclusion during execution. For example, if two programs running at the same time need to access the same variable, there is a need for mutual exclusion. In particular, if *prog_a* controls arm[1] and *prog_b* controls arm[2] and both arms need to move to the same destination, mutual exclusion of the destination is required to avoid a crash.

SEMAPHORE is a PDL2 data type consisting of an integer counter that is incremented each time a signal occurs and decremented each time a wait occurs and a queue including all the programs waiting on that semaphore.

Synchronization is done using the WAIT (not to be confused with WAIT FOR) and SIGNAL statements.

The syntax is as follows:

```
WAIT semaphore_var
SIGNAL semaphore_var
```

When a program wants a resource, it uses the WAIT statement. When the program finishes with the resource it uses the SIGNAL statement. If another program is waiting for that resource, that program can resume executing. Synchronization is done on a first in first out basis.

Since PDL2 semaphores count, it is important to have a matching number of SIGNALs and WAITS. Too many signals will violate mutual exclusion of the resource (unless multiple instances of the resource exist). For example, when programs *badsem1* and *badsem2* outlined below are activated, the mutual exclusion will function properly for the first cycle. However, upon the second cycle of the programs, we lose mutual exclusion because there is one too many SIGNALs in *badsem1*.

If the first SIGNAL statement of *badsem1* were above the CYCLE statement, all would be ok for the following cycles.

```

PROGRAM badsem1
VAR resource : SEMAPHORE EXPORTED FROM badsem1
BEGIN
CYCLE
    SIGNAL resource --to initialize semaphore
    .
    .
    WAIT resource
    -- mutual exclusive area
    SIGNAL resource
    .
    .
END badsem1

PROGRAM badsem2
VAR resource : SEMAPHORE EXPORTED FROM badsem1
BEGIN
CYCLE
    .
    .
    WAIT resource
    -- mutual exclusive area
    SIGNAL resource
    .
    .
END badsem2

```

Another situation to avoid is called a deadlock. A deadlock occurs when all of the programs are waiting for the resource at the same time. This means none of them can signal to let a program go. For example, if there is only one program and it starts out waiting for a resource, it will be deadlocked.

It is also important to reset semaphores when execution begins. This is done with the CANCEL semaphore statement. The syntax is as follows:

```
CANCEL semaphore_var
```

The CANCEL semaphore statement resets the signal counter to 0. It results in an error if programs are currently waiting on the SEMAPHORE variable. This statement is useful for clearing out unused signals from a previous execution.

6.4 Program Scheduling

Program scheduling is done on a priority basis. The programmer can set a program priority using the PRIORITY attribute on the PROGRAM statement.

```
PROGRAM test PRIORITY=1
```

Valid priority values range from 1-3, with 3 being the highest priority and 2 being the default.

If two or more programs have equal priority, scheduling is done on a round robin basis, based on executing a particular number of statements.

Programs that are suspended, for example those waiting on a READ or SEMAPHORE or paused programs, are not included in the scheduling.

Interrupt service routines use the same arm, priority, stack, program-specific predefined variable values, and trapped errors as the program which they are interrupting.

7. ROUTINES

The following information are available about routines:

- Procedures and Functions
- Parameters
- Declarations
- Passing Arguments

A ROUTINE is structured like a program, although it usually performs a single task. Routines are useful to shorten and simplify the main program. Tasks that are repeated throughout a program or are common to different programs can be isolated in individual routines.

A program can call more than one routine and can call the same routine more than once. When a program calls a routine, program control is transferred to the routine and the statements in the routine are executed. After the routine has been executed, control is returned to the point from where the routine was called. Routines can be called from anywhere within the executable section of a program or routine.

The programmer writes routines in the declaration section of a program. As shown in the following example, the structure of a routine is similar to the structure of a program.

```

PROGRAM arm_check
-- checks three arm positions and digital outputs
VAR
    perch, checkpt1, checkpt2, checkpt3 : POSITION

ROUTINE reset_all -- routine declaration
-- resets outputs to off and returns arm to perch
VAR
    n : INTEGER
BEGIN
    FOR n := 21 TO 23 DO
        $DOUT[n] := OFF
    ENDFOR
    MOVE TO perch
END reset_all

BEGIN -- main program
    reset_all -- routine call
    MOVE TO checkpt1
    $DOUT[21] := ON
    reset_all
    MOVE TO checkpt2
    $DOUT[22] := ON
    reset_all
    MOVE TO checkpt3
    $DOUT[23] := ON
    reset_all
END arm_check

```

PDL2 also includes built-in routines. These are predefined routines that perform

commonly needed tasks.

They are listed and described in [Chap.11. - BUILT-IN Routines List](#).

7.1 Procedures and Functions

The preceding example shows a procedure routine. A procedure routine is a sequence of statements that is called and executed as if it were a single executable statement.

Another kind of routine is a function routine. A function routine is a sequence of computations that results in a single value being returned. It is called and executed from within an expression.

Function routines often are used as part of a control test in a loop or conditional branch or as part of an expression on the right side of an assignment statement. The following example shows a function routine that is used as the test in an IF statement.

```

PROGRAM input_check

ROUTINE time_out : BOOLEAN
-- checks to see if input is received within time limit
CONST
    time_limit = 3000
VAR
    time_slice : INTEGER
BEGIN
    $TIMER[1] := 0
    REPEAT
        time_slice := $TIMER[1]
    UNTIL ($DIN[1] = ON) OR (time_slice > time_limit)
    IF time_slice > time_limit THEN
        RETURN (TRUE)
    ELSE
        RETURN (FALSE)
    ENDIF
END time_out

BEGIN -- main program
    .
    .
    IF time_out THEN
        WRITE (Timeout Occurred)
    ENDIF
    .
END input_check

```

7.2 Parameters

To make routines more general for use in different circumstances, the programmer can use routine parameters rather than using constants or program variables directly. Parameters are declared as part of the routine declaration. When the routine is called, data is passed to the routine parameters from a list of arguments in the routine call. The number of arguments passed into a routine must equal the number of parameters declared for the routine. Arguments also must match the declared data type of the parameters.

The following example shows how the *time_out* routine can be made more general by

including a parameter for the input signal index. The modified routine can be used for checking any input signal.

```

PROGRAM input_check

ROUTINE time_out (input : INTEGER) : BOOLEAN
-- checks to see if input is received within time limit
CONST
  time_limit = 3000
VAR
  time_slice : INTEGER
BEGIN
  $TIMER[1] := 0
  REPEAT
    time_slice := $TIMER[1]
    UNTIL ($DIN[input] = ON) OR (time_slice > time_limit)
    IF time_slice > time_limit THEN
      RETURN (TRUE)
    ELSE
      RETURN (FALSE)
    ENDIF
  END time_out

  BEGIN -- main program
  .
  .
  .
  IF time_out(6) THEN
    WRITE (Timeout Occurred)
  ENDIF
  .
  .
END input_check

```

7.3 Declarations

PDL2 uses two different methods of declaring a routine, depending on whether the routine is a procedure or function.

It is also allowed to implement Shared Routines.

The following topics are fully described in current paragraph:

- Declaring a Routine
- Parameter List
- Constant and Variable Declarations
- Function Return Type
- RETURN Statement
- Shared Routines

7.3.1 Declaring a Routine

- Procedure
- Function

7.3.1.1 Procedure

The syntax of a procedure declaration is as follows:

```
ROUTINE proc_name <parameter_list>
<constant and variable declarations>
BEGIN
    <statement...>
END proc_name
```

7.3.1.2 Function

The syntax of a function declaration includes a return data type and must include at least one RETURN statement that will be executed, as follows:

```
ROUTINE func_name <parameter_list> : return_type
<constant and variable declarations>
BEGIN
    <statement...> must include RETURN statement
END func_name
```

7.3.2 Parameter List

The optional *parameter_list* allows the programmer to specify which data items are to be passed to the routine from the calling program or routine.

The syntax of the parameter list is as follows:

```
<(id<, id>... : id_type<; id<, id>... : id_type>...)>
```

The *id_type* can be any PDL2 data type with the following restrictions:

- for a STRING, a size cannot be specified. This is so that a STRING of any size can be used;
- for a JOINTPOS or XTNDPOS, an arm number cannot be specified. This is so that a value for any arm can be used;
- for an ARRAY, a size cannot be specified. This is so that an ARRAY of any size can be used. An asterisk (*) may be used to indicate a one-dimension ARRAY but it is not required. For example:

```
part_dim : ARRAY OF INTEGER
part_dim : ARRAY[*] OF INTEGER
```

For a two-dimension ARRAY, asterisks (*,*) must be included as follows:

```
part_grid : ARRAY[*,*] OF INTEGER
```

7.3.3 Constant and Variable Declarations

Routines can use VAR and CONST declarations in the same way the main program does, with the following exceptions:

- PATH and SEMAPHORE variables are not allowed;
- EXPORTED FROM clauses are not allowed;
- attributes are not allowed;
- NOSAVE clauses are not allowed.

Variables and constants declared in the VAR and CONST section of a routine are called local to the routine. This means they have meaning only within the routine itself. They cannot be used in other parts of the program. In addition, identifiers declared local to a routine cannot have the same name as the routine.

Variables and constants declared in the VAR and CONST section of the main program can be used anywhere in the program, including within routines. They belong to the main program context. NOTE that if a variable declared inside a routine has the same name as a variable declared at the main program level, the routine will access the locally declared variable.

For example:

```

PROGRAM example
VAR
  x : INTEGER

ROUTINE test_a
VAR
  x : INTEGER
BEGIN
  x := 4
  WRITE(Inside test_a x = , x, NL)
END test_a
ROUTINE test_b
BEGIN
  WRITE(Inside test_b x = , x, NL)
END test_b

BEGIN
  x := 10
  WRITE(The initial value of x is , x, NL)
  test_a
  WRITE(After test_a is called x = , x, NL)
  test_b
  WRITE(After test_b is called x = , x, NL)
END example
  
```

Results of the output:

```

The initial value of x is 10
Inside test_a x = 4
After test_a is called x = 10
Inside test_b x = 10
After test_b is called x = 10
  
```

The distinction of belonging to the main program context versus local to a routine, is referred to as the scope of a data item.

7.3.3.1 Stack Size

The only limit to routine calls is the size of the program stack, since each call (including an interrupt service routine call) takes up a portion of this stack. The amount of stack used for each routine call is proportional to the number of local variables declared in the routine. The stack size can be specified by the STACK attribute on the PROGRAM statement.

7.3.4 Function Return Type

The *return_type* for a function declaration can be any PDL2 data type with the following restrictions:

- for a STRING, a size cannot be specified. This is so that a STRING of any size can be used;
- for a JOINTPOS or XTNDPOS, an arm number cannot be specified. This is so that a value for any arm can be used;
- for an ARRAY, a size cannot be specified. This is so that an ARRAY of any size can be used. An asterisk (*) may be used to indicate a one-dimension ARRAY but it is not required.
- For example:

part_dim : **ARRAY OF INTEGER**

part_dim : **ARRAY[*] OF INTEGER**

- for a two-dimension ARRAY, asterisks (*,*) must be included as follows:

part_grid : **ARRAY[*,*] OF INTEGER**

- the PATH and SEMAPHORE data types cannot be used.

7.3.4.1 Functions as procedures

It is allowed to call a function without assigning its result to a variable. This means that a function can be used in all contexts and when it is called in the context of a procedure, it does not need to return any value.

7.3.5 RETURN Statement

The RETURN statement is used to return program control from the routine currently being executed to the place where it was called.

For function routines, it also returns a value to the calling program or routine.

The syntax for a RETURN statement is as follows:

```
RETURN <(value)>
```

The RETURN statement, with a return value, is required for functions. The value must be an expression that matches the return data type of the function. If the program interpreter does not find a RETURN while executing a function, an error will occur when the interpreter reaches the END statement.

The RETURN statement can also be used in procedures. RETURN statements used in procedures cannot return a value. If a RETURN is not used in a procedure, the END statement transfers control back to the calling point.

7.3.6 Shared Routines

Routines can be declared as owned by another program (by using the optional EXPORTED FROM clause) or as public to be used by other programs (by using the optional EXPORTED FROM clause together with or without the attribute).

- The syntax for declaring a routine owned by another program, is as follows:

```
EXPORTED FROM prog_name
```

Prog_name indicates the name of the external program owning the specified routine. For example:

```
PROGRAM pippo
.
.
ROUTINE error_check EXPORTED FROM utilities
```

- There are two different syntaxes for declaring a routine to be public for use by other programs:

- EXPORTED FROM *prog_name*

prog_name indicates the name of the current program, which owns the specified routine.

For example:

```
PROGRAM utilities
ROUTINE error_check EXPORTED FROM utilities

PROGRAM prog_1
ROUTINE error_check EXPORTED FROM utilities -- error_check routine
-- is imported from program utilities
```

- EXPORTED FROM *prog_name*

prog_name indicates the name of the current program, which owns the specified routine. This routine can be imported by another program, by means of the [IMPORT Statement](#).

For example:

```
PROGRAM utilities
.
.
ROUTINE error_check EXPORTED FROM utilities

PROGRAM prog_1
IMPORT `utilities` -- any variables and routines
-- are imported from program utilities,
-- including error_check routine
```

The declaration and executable sections of the routine appear only in the program that owns the routine.

Refer to [par. 3.2.4 Shared types, variables and routines on page 67](#) for more information.

7.4 Passing Arguments

Arguments passed to a routine through a routine call must match the parameter list in the routine declaration in number and type. An argument can be any valid expression that meets the requirements of the corresponding parameter.

There are two ways to pass arguments to parameters:

- [Passing By Reference](#);
- [Passing By Value](#);
- [Optional parameters](#);

- Variable arguments (Varargs);
- Argument identifier.

7.4.1 Passing By Reference

When an argument is passed by reference, the corresponding parameter has direct access to the contents of that argument. Any changes made to the parameter inside of the routine will also change the value of the argument. To pass by reference, the argument must refer to a user-defined variable.



Note that a variable which has been declared with the **CONST attribute cannot be passed by reference to a routine!**

7.4.2 Passing By Value

When an argument is passed by value, only a copy of that value is sent to the routine. This means that any changes made to the parameter inside of the routine do not affect the value of the argument. To pass a user-defined variable by value, enclose the argument in parentheses.

Arguments that are constants, expressions, or literals automatically will be passed by value, as will any predefined variable (including port arrays), or any INTEGER variable that is passed to a REAL parameter. Other variables, however, are passed by reference unless enclosed in parentheses.

Example of a variable being passed:

```
check_status (status)      -- passed by reference
check_status ( (status) )  -- passed by value
```



The routine itself is executed in the same manner regardless of how the arguments were passed to it

The following is an example of routines that use pass by reference and pass by value:

```
PROGRAM example
VAR
    x : INTEGER

ROUTINE test_a(param1 : INTEGER)
BEGIN
    param1 := param1 + 4
    WRITE(Inside test_a param1 = , param1, NL)
END test_a

BEGIN
    x := 10
    WRITE(The initial value of x is , x, NL)
    test_a((x))  -- pass by value
    WRITE(After pass by value x = , x, NL)
    test_a(x)    -- pass by reference
    WRITE(After pass by reference x = , x, NL)
```

```
END example
```

Results of the output:

```
The initial value of x is 10
Inside test_a param1 = 14
After pass by value x = 10
Inside test_a param1 = 14
After pass by reference x = 14
```

7.4.3 Optional parameters

It is allowed to declare routines with optional arguments: they can be optionally set when the routine is called.

Example of specifying optional variables when declaring a routine:

```
ROUTINE aa (i1:INTEGER; r2:REAL(); s3:STRING('hello'))
```

In the shown above example, **i1** is required; **r2** and **s3** are optional arguments: this is indicated by the brackets which follow the datatype specification.

When a value is declared within the brackets, it means that such a value will be the default value when the routine is called. If no values are declared, the default value at calling time will be UNINIT.



NOTEs:

- the values for optional parameters, when not passed as an argument, are uninitialized except for arrays in which the dimensions are 0. For jointpos and xtndpos the program's executing Arm is used to initialize the value;
- the optional values in all declarations of the routines must be the same unless the optional value is not defined in the EXPORTED FROM Clause;
- optional arguments can be used with **CALLS Statement**;
- optional arguments can be specified in Interrupt Service Routines (ISR); note that value is setup when the routine is called.

7.4.4 Variable arguments (Varargs)

It is allowed to declare routines with a variable number of arguments: they are passed when the routine is called.

At declaration time, neither the total amount of arguments, nor their data types are to be specified.

Variable arguments are marked in the routine declaration by 3 dots after the last declared variable.

Examples:

```
ROUTINE rx(ai_value:INTEGER; ...) --ai_value is required
ROUTINE bb(...) -- no required arguments
```

When the routine is called, up to 16 additional arguments of any datatype (excluding PATH, NODE, SEMAPHORE and ARRAY of these) can be supplied to the routine.

Example:

```
r2 (5, mypos)
```

The user is also allowed to pass [Optional parameters](#).

Example:

```
r2(5, mypos, , 6)
```

To handle the information about the variable arguments, inside the routine, some specific built-ins are available:

- [ARG_COUNT Built-In Function](#) - to get the total amount of arguments supplied to the routine
- [ARG_INFO Built-In Function](#) - to get information about the indexed argument
- [ARG_GET_VALUE Built-in Procedure](#) - to obtain the value of the indexed argument
- [ARG_SET_VALUE Built-in Procedure](#) - to set a value to the indexed argument.

NOTES:

- [Optional parameters](#) can be passed in Varargs routines; in this case the returned datatype from ARG_INFO is 0;
- the argument can be an ARRAY;
- Varargs routines can be used in Interrupt Service Routines and in [CALLS Statement](#), but only the defined arguments;
- [Argument identifiers](#) cannot be used as argument switch (because there is no identifiers!) in Varargs routines.

7.4.5 Argument identifier

When calling a routine, it is allowed to use argument identifiers which have been specified at declaration time.

Provided that the user is now allowed to use [Optional parameters](#), when calling a routine it is needed to be able to specify which is the argument the passed value is referred to. This is possible by means of using the argument identifier. It must be preceded by '/' (slash) and followed by '=', as shown in the below examples.

Examples:

```
ROUTINE r1(par1:INTEGER; par2, timeout:INTEGER(); rr:REAL())
...
...
r1(1, /timeout=5, 5.2) --par2 is not passed; timeout is assigned
--a value of 5; rr is assigned 5.2
r1(1, , 5.2) --par1 gets a value of 1; par2 gets no values;
--timeout gets a value of 1; rr gets 5.2
r1(2, 10, /rr=3.7) --par1 gets a value of 2; par2 gets 10;
--timeout is not passed; rr gets 3.7
```

NOTES:

- all parameters preceding this argument must have been provided or must be optional;
- [argument identifiers](#) cannot be used in Interrupt Service Routines.

8. CONDITION HANDLERS

Condition handlers allow a program to monitor specified conditions in parallel with normal execution and, if the conditions occur, take corresponding actions in response. Typical uses include monitoring and controlling peripheral equipment and handling errors.

For example, the following condition handler checks for a PAUSE condition. When the program is paused, the digital output \$DOUT[21] is turned off. This example might be used to signal a tool to turn off if a program is paused unexpectedly.

```

CONDITION [1] :                                -- defines condition handler
  WHEN PAUSE DO
    $DOUT [21] := OFF
  ENDCONDITION

ENABLE CONDITION [1]                            -- enables condition handler

```

The following information are available:

- [Defining Condition Handlers](#)
- [Enabling, Disabling, and Purging](#)
- [Variable References](#)
- [Conditions](#)
- [Actions](#)

8.1 Operations

As shown in the previous example, using condition handlers is a two-step process. First the condition handler is defined, and then it can be enabled. Condition handlers can also be disabled and purged.

- [Defining Condition Handlers](#)
- [Enabling, Disabling, and Purging](#)

8.1.1 Defining Condition Handlers

Condition handlers are defined in the executable section of a program using the CONDITION statement

The syntax of the CONDITION statement is as follows:

```

CONDITION [number] <FOR ARM [n] > <NODISABLE> <ATTACH> <SCAN (number) >:
  WHEN cond_expr DO
    action_list
  <WHEN cond_expr DO
    action_list>...
ENDCONDITION

```

The Programmer identifies each condition handler by a number, an INTEGER expression that can range from 1 to 255.

Conditions that are to be monitored are specified in the condition expression

(*cond_expr*). Multiple conditions can be combined into a single condition expression using the BOOLEAN operators AND and OR. [Conditions](#) that can be monitored are explained later in this chapter. When the expression becomes TRUE, the condition handler is said to be triggered.

The action list (*action_list*) specifies the actions that are to be taken when the condition handler is triggered. The [Actions](#) that can be included in an action list are explained later in this chapter.

Multiple WHEN clauses can be included in a single condition handler definition. The effect is to cause the individual WHEN clauses to be enabled together and disabled together. This also makes the clauses mutually exclusive, meaning only one clause will be triggered from the set.

- [FOR ARM Clause](#)
- [NODISABLE Clause](#)
- [ATTACH Clause](#)
- [SCAN Clause](#)

8.1.1.1 FOR ARM Clause

The optional FOR ARM clause can be used to designate a particular arm for the condition handler. If a condition has the FOR ARM clause specified, a motion event will trigger that arm. Similarly, any arm-related actions will apply to the arm specified by FOR ARM. If the FOR ARM is not specified, the default arm (PROG_ARM or \$DFT_ARM) will be used for any arm-related actions. Globally enabled motion events will apply to any moving arm while local events will apply to the arm of the MOVE statement they are associated with (via the WITH option).

For example:

```

PROGRAM ch_test PROG_ARM=1
.
.
BEGIN
    CONDITION[23] FOR ARM[2] :
        WHEN DISTANCE 60 BEFORE END DO
            LOCK
        WHEN $ARM_DATA[4].PROG_SPD_OVR > 50 DO
            LOCK ARM[4]
    ENDCONDITON

    CONDITION[24] :
        WHEN DISTANCE 60 BEFORE END DO
            LOCK
    ENDCONDITON

```

In CONDITION[23], the first WHEN clause and the LOCK apply to arm 2 as designated by the FOR ARM clause. The second WHEN and LOCK apply to arm 4, as explicitly designated in the condition and action.

In CONDITION[24], WHEN and LOCK clauses apply to the arm of the MOVE statement to which the condition is associated.

8.1.1.2 NODISABLE Clause

The optional NODISABLE clause indicates the condition handler will not automatically

be disabled when the condition handler is triggered.

The NODISABLE clause is not allowed on condition handlers that contain state conditions.

8.1.1.3 ATTACH Clause

The optional ATTACH clause causes the condition handler to be attached immediately after it is defined. If not specified, the condition handler can be attached elsewhere in the program using the ATTACH CONDITION statement.

The syntax for the ATTACH CONDITION statement is as follows:

```
ATTACH CONDITION [number] <, CONDITION [number]> ...
```

When a condition handler is attached, only code belonging (in terms of context program) to the same program containing the ATTACH can enable, disable, purge, detach, or redefine the condition handler. The syntax is as follows:

```
DETACH CONDITION || ALL | [number] <, CONDITION [number]> ... ||
```

All attached condition handlers are detached when the ALL option is used. Otherwise, only those specified are detached.

For more information, refer to the description of the ATTACH & DETACH statements in [Condition Handlers](#) chapter.

8.1.1.4 SCAN Clause

The optional SCAN clause can be used for indicating a different rate for the scanning of the condition handler.

If this attribute is not specified, the condition handler is monitored accordingly to the scan time defined in the system (\$TUNE[1] value). If the SCAN clause is present in the condition definition, the expressions that are states in such condition handler will be scanned less frequently.

A positive INTEGER variable or expression must be specified in round brackets near the SCAN clause; this number indicates the multiplicator factor to apply to the regular scan rate during the condition handler monitoring.

For example:

```
VAR a, b: INTEGER

CONDITION [55] SCAN (2) :
  WHEN a = b DO
    ENABLE CONDITION [4]
  ENDCONDITION
```

If the regular scan time of condition handlers is 20 milliseconds (default value for \$TUNE[1]), the above defined condition is monitored every 40 milliseconds.

The SCAN clause is only useful if applied to conditions that are states and that do not require to be scanned very frequently. The SCAN clause acts as a filter for reducing the system overhead during the monitoring.

Note that, when the condition triggers, it is automatically disabled. Either use the NODISABLE clause (trapping on the error 40016) or the ENABLE CONDITION statement for maintaining the condition always enabled.

Events are not influenced by the SCAN clause.

Refer to other sections of this chapter for the definitions of states and events.

8.1.2 Enabling, Disabling, and Purging

A condition expression is monitored only when the condition handler is enabled. Condition handlers can be globally enabled using the ENABLE CONDITION statement or action. The syntax is as follows:

```
ENABLE CONDITION [number] <, CONDITION [number]> ...
```

Condition handlers can be temporarily enabled as part of a MOVE statement WITH clause. The syntax is as follows:

```
WITH CONDITION [number] <, CONDITION [number]> ...
```

For example:

```
PROGRAM example
.
.
.
BEGIN
    CONDITION[1]:
        WHEN AT START DO
            $DOUT[22] := ON
    ENDCONDITION
.
.
.
MOVE TO p1 WITH CONDITION[1]
.
.
.
END example
```

Condition handlers are disabled automatically when they are triggered (unless the NODISABLE clause is specified).

Globally enabled condition handlers can be disabled using the DISABLE CONDITION statement or action. The syntax is as follows:

```
DISABLE CONDITION || ALL | [number] <, CONDITION [number]> ... ||
```

All enabled condition handlers are disabled when the ALL option is used. Otherwise, only those specified are disabled.

If a condition handler is also currently enabled as part of a WITH clause, it will be disabled when the move finishes, not when the DISABLE CONDITION statement is executed.

Condition handlers that are temporarily enabled as part of a WITH clause are automatically disabled when the motion is completed or canceled. They are also automatically disabled when the motion is suspended (for example, by a LOCK statement) and re-enabled when the motion is resumed.

COND_ENABLED and COND_ENBL_ALL built-ins can be used for testing if a certain condition is enabled (globally or locally). Refer to [BUILT-IN Routines List](#) chapter for further details.

Condition handler definitions are automatically purged when the program is deactivated. Condition handler definitions can also be purged using the PURGE CONDITION statement or action. Purged condition handlers cannot be re-enabled. The syntax is as follows:

```
PURGE CONDITION || ALL | [number] <, CONDITION [number]> ... ||
```

All defined condition handlers are purged when the ALL option is used. Otherwise, only those specified are purged.

If a condition handler is currently enabled as part of a WITH clause, it cannot be purged.

8.2 Variable References

The types of variables that can be referenced within a condition handler is restricted to variables belonging to the main program context or predefined variables. Local variables or routine parameters are not allowed in a condition handler unless their value is obtained at the time the condition handler is defined. If an array element is referenced in a condition handler, the array index is evaluated at condition definition time. This must be considered when using variables as array indexes in conditions. In addition, predefined variables that require the use of built-in routines for access are not allowed within a condition handler. Also, the value of a predefined variable that is limit checked can not be modified.

8.3 Conditions

A condition can be a state or an event. States remain satisfied as long as the state exists. They are monitored at fixed intervals by a process called scanning. Events are satisfied only at the instant the event occurs. Consequently, event conditions cannot be joined by the AND operator in a condition expression.

The user should take care of the way in which his conditions are structured. Events must be preferred to states; if a state, a condition should remain enabled only when needed, because it is polled each scan time (10 or 20 ms); in case of events, the NODISABLE clause is preferred to an ENABLE CONDITION action.

Multiple conditions can be combined into a single condition expression using the BOOLEAN operators AND and OR. The AND operator has a higher precedence which means the condition expression

`$DIN[1] OR $DIN[2] AND $DIN[3]`

is triggered when \$DIN[1] is ON or when \$DIN[2] and \$DIN[3] are both ON. An error occurs if BOOLEAN state conditions are combined with the AND or OR operators grouped in parentheses. For example, `($DIN[1] OR $DIN[2]) AND $DIN[3]` will produce an error.

See [Chap.12. - Predefined Variables List](#) for the description of \$THRD_CEXP and \$THRD_PARAM variables that can be helpful when programming with conditions.

The following conditions can be included in a condition expression:

- [RELATIONAL States](#)
- [BOOLEAN State](#)
- [DIGITAL PORT States](#)
- [DIGITAL PORT Events](#)
- [SYSTEM Events](#)
- [USER Events](#)
- [ERROR Events](#)
- [PROGRAM Events](#)
- [Event on Cancellation of a Suspendable Statement](#)

- MOTION Events



Please note that if the involved variables in states or events are uninitialized, no error occurs.

8.3.1 RELATIONAL States

A relational state condition tests the relationship between two operands during every scan. The condition is satisfied when the relationship is TRUE. Any relational operator can be used. The operands can be user-defined variables, predefined variables, port arrays, literal values, or constants. At least one operand must be a variable. When used in a TIL or WHEN clause, the operands cannot be local variables or parameters.

Using REAL operands in a condition expression can un-necessarily slow down the system, which can cause errors to be generated. This becomes a bigger problem when the system does not contain a floating point processor, or multiple condition expressions use REAL operands. To avoid this problem, use INTEGER operands wherever possible. If a REAL operand must be used, simplify the expression as much as possible.

The variables used in relational state conditions do not have to be initialized when the condition handler is defined. In addition, tests for uninitialized variables are not performed when the condition handler is scanned.

For example:

```
WHEN bool_var = TRUE DO
WHEN int_var <= 10 DO
WHEN real_var > 10.0 DO
WHEN $TIMER[1] > max_time DO
WHEN $GIN[2] <> $GIN[3] DO
```

8.3.2 BOOLEAN State

A BOOLEAN state condition tests a BOOLEAN variable during every scan. The condition is satisfied when the variable is TRUE. When used in a TIL or WHEN clause, the BOOLEAN variable cannot be a local variable or parameter.

For example:

```
WHEN bool_var DO
```

The BOOLEAN operator NOT can be used to specify the condition is satisfied when the variable is FALSE. For example:

```
WHEN NOT bool_var DO
```

bool_var can also be the result of the BIT_TEST built-in function. This function tests whether a specified bit of an INTEGER is ON or OFF. The first argument is the variable to be tested, the second argument indicates the bit to be tested, and the third argument indicates whether the bit is to be tested for ON or OFF. When used in a condition expression, the third argument of BIT_TEST must be specified. In addition, variables used as arguments must not be local or parameters. (Refer to the BIT_TEST section of Chap.11. - BUILT-IN Routines List for additional information on BIT_TEST.) The following is an example using BIT_TEST in a condition expression:

```
WHEN BIT_TEST (int_var, bit_num, TRUE) DO
```

When combining two BOOLEAN variable state conditions with the AND or OR operator, do not use parentheses around the expression. If parentheses are used, the editor treats the expression operator as an equal (=) instead of AND or OR. The following is an example of the proper syntax to use:

```
WHEN a AND b DO
```

8.3.3 DIGITAL PORT States

A digital port state condition monitors one of the digital I/O port array signals. The signal is tested during every scan. The condition is satisfied if the signal is on during the scan.

For example:

```
WHEN $DIN[1] DO
WHEN $DOUT[22] DO
```

The BOOLEAN operator NOT can be used to specify the condition is satisfied if the signal is OFF during the scan. For example:

```
WHEN NOT $DIN[1] DO
```

8.3.4 DIGITAL PORT Events

A digital port event condition also monitors one of the digital I/O port array signals. The \$DIN, \$DOUT, \$IN, \$OUT, \$SDI, \$SDO, and \$BIT port arrays can be monitored as events. \$FDIN cannot be monitored as an event.

For events, the initial signal value is tested when the condition handler is enabled. Each scan tests for the specified change in the signal. The condition is satisfied if the change occurs while the condition handler is enabled.

The signal change is specified as follows:

- + (changes from OFF to ON)
- (changes from ON to OFF)

For example:

```
WHEN $DIN[1]+ DO
WHEN $DOUT[22]- DO
```

The BIT_FLIP built-in function can be used for monitoring the event of positive or negative transition of a bit inside one of the following analogue port arrays : \$AIN, \$AOUT, \$GIN, \$GOUT, \$WORD, \$USER_BYTE, \$USER_WORD, \$USER_LONG, \$PROG_UBYTE, \$PROG_UWORD, \$PROG ULONG. For further details about this function, please refer to [Chap.11. - BUILT-IN Routines List](#).

8.3.5 SYSTEM Events

A system event condition monitors a system generated event. The condition is satisfied when the specified event occurs while the condition handler is enabled. The condition expression is scanned only when a system event occurs.

System events include the following:

- **POWERUP:** resumption of program execution after a power fail recovery;
- **HOLD:** each press of the HOLD button or execution of the PDL2 statement;
- **START:** each press of the START button;

- **EVENT code:** occurrence of a system event, identified by code;
- **SEGMENT WAIT path_var:** \$SEG_WAIT field of the executed node is TRUE;
- **WINDOW SELECT scrn_num:** selection of a different input window on the screen.

The POWERUP condition is triggered at the start of power failure recovery. A small delay may be necessary before procedures such as DRIVE ON can be performed. It is not sufficient to test the POWERUP condition for understanding that the system has completely recovered from power failure. Please refer to [Chap.13. - Power Failure Recovery](#) which contains several sample program demonstrating power failure recovery techniques.

The HOLD condition is triggered by either pressing the HOLD button or by executing the PDL2 HOLD statement. The HOLD condition is also triggered when a power failure occurs. The START button must be pressed at least once before the event can be triggered by pressing the HOLD button. The HOLD button that is pressed must be on the currently enabled device. In AUTO mode, it must be the HOLD button on the TP. When in REMOTE mode, the HOLD signal must be used to trigger this event. In either mode, execution of the HOLD statement will cause the event to trigger. This event should not be used in PROGR mode.

The START condition is triggered when START button is pressed. This event can only be used in AUTO or REMOTE mode. The pressed START button must be on the current enabled device. In AUTO mode, it must be the TP START button; in REMOTE mode, the START signal must be used to trigger such an event.

Predefined constants are available for some of the event codes. The following is a list of valid codes for use with the EVENT condition.

EVENT Code	Meaning
80	Triggered when WinC5G Program connects to the Control Unit.
81	Triggered when WinC5G Program disconnects from the Controller Unit or an error occurs.
82	reserved
83	Triggered when the A1 softkey on the TP Virtual Keyboard is pressed.
84	Triggered when the A2 softkey on the TP Virtual Keyboard is pressed.
85	Triggered when the general override is changed by pressing the TP % key or by issuing the SET ARM GEN_OVR command
91	U1 softkey is pressed (TP right menu)
92	U2 softkey is pressed (TP right menu)
93	U3 softkey is pressed (TP right menu)
94	U4 softkey is pressed (TP right menu)
99	Triggered when the DRIVES are turned ON
100	Triggered when the DRIVES are turned OFF
101	Modal selector switch turned to T1
102	Modal selector switch turned to AUTO
103	Modal selector switch turned to REMOTE
106	Transition of the system into PROGR state

EVENT Code	Meaning
107	Transition of the system into AUTO state
108	Transition of the system into REMOTE state
109	Transition of the system into ALARM state
110	Transition of the system into HOLD state
111	TP enabling device switch pressed
112	TP enabling device switch released
113	reserved
114	reserved
115	TP is disconnected
116	TP is connected
117	Operations Panel key switch turned to T2
118	AUTO-T state is entered
119	119 Triggered when the default arm for jogging is selected
120	reserved
125	Arm 1 is ready to receive offsets from an external sensor
126	Arm 2 is ready to receive offsets from an external sensor
127	Arm 3 is ready to receive offsets from an external sensor
128	Arm 4 is ready to receive offsets from an external sensor
129	^C Key pressure from TP or video/keyboard of PC (with WinC5G active)
130	Triggers when Arm 1 enters in the automatic locked state (enabled setting \$RCVR_LOCK to TRUE)
131	Triggers when Arm 2 enters in the automatic locked state (enabled setting \$RCVR_LOCK to TRUE)
132	Triggers when Arm 3 enters in the automatic locked state (enabled setting \$RCVR_LOCK to TRUE)
133	Triggers when Arm 4 enters in the automatic locked state (enabled setting \$RCVR_LOCK to TRUE)
134	Input to the sphere defined in \$ON_POS_TBL[1]
135	Input to the sphere defined in \$ON_POS_TBL[2]
136	Input to the sphere defined in \$ON_POS_TBL[3]
137	Input to the sphere defined in \$ON_POS_TBL[4]
138	Input to the sphere defined in \$ON_POS_TBL[5]
139	Input to the sphere defined in \$ON_POS_TBL[6]
140	Input to the sphere defined in \$ON_POS_TBL[7]
141	Input to the sphere defined in \$ON_POS_TBL[8]
142	Output from the sphere defined in \$ON_POS_TBL[1]
143	Output from the sphere defined in \$ON_POS_TBL[2]
144	Output from the sphere defined in \$ON_POS_TBL[3]

EVENT Code	Meaning
145	Output from the sphere defined in \$ON_POS_TBL[4]
146	Output from the sphere defined in \$ON_POS_TBL[5]
147	Output from the sphere defined in \$ON_POS_TBL[6]
148	Output from the sphere defined in \$ON_POS_TBL[7]
149	Output from the sphere defined in \$ON_POS_TBL[8]
154	Foreign Language change
155	A USB flash drive has been connected into the XD: USB port
156	A USB flash drive has been disconnected from the XD: USB port
157	A forward movement in progress
158	A backward motion in progress
159	A jog motion in progress
160	A manual motion just stopped
161	Either an alarm has occurred requesting a confirmation (latch) or a confirmation has been given to an alarm requesting it
177	A USB flash drive has been connected into the TX: USB port, on the Teach Pendant
178	A USB flash drive has been disconnected from the TX: USB port, on the Teach Pendant
179	An I/O port has been forced
180	An I/O port has been unforced
181	reserved
182	reserved
183	New log message
187	TP START pressed
188	TP START released
189	TP HOLD pressed
195	New VP2 Screen defined
196	VP2 Screen deleted
197	Collision detection occurred
198	Screen added - specific per session
199	Screen is now available for the session
200	Screen removed - specific per session
201	Screen set - specific per session
204	Language changed
207	Login performed
208	Logout performed
209	Transition of HDIN[1] channel 1 of Arm 1
210	Transition of HDIN[2] channel 1 of Arm 2
211	Transition of HDIN[3] channel 1 of Arm 3
212	Transition of HDIN[4] channel 1 of Arm 4

EVENT Code	Meaning
213	Transition of HDIN[5] channel 2 of Arm 1
214	Transition of HDIN[6] channel 2 of Arm 2
215	Transition of HDIN[7] channel 2 of Arm 3
216	Transition of HDIN[8] channel 2 of Arm 4

The use of events related to HDIN (209..216) should be preceeded by [HDIN_SET Built-In Procedure](#) call so that the monitoring of \$HDIN transition is enabled.

The SEGMENT WAIT condition is triggered before the motion to a path node whose \$SEG_WAIT field is TRUE. At this time the processing of the path is suspended until a SIGNAL SEGMENT statement or action is executed for that path.

The WINDOW SELECT condition is triggered when a different window is selected for input on the specified screen. A window is selected for input either by the SEL key on the keyboard or TP or by the WIN_SEL built-in routine. After the condition triggers, the SCRn_GET built-in should be used to determine which window has been selected. Refer to [Chap.11. - BUILT-IN Routines List](#) for more information about WIN_SEL and SCRn_GET.

For example:

```
WHEN POWERUP DO
WHEN HOLD DO
WHEN START DO
WHEN EVENT AE_CALL DO
WHEN SEGMENT WAIT weld_path DO
WHEN WINDOW SELECT SCRn_USER DO
```

8.3.6 USER Events

This class of events can be used as a means of programs synchronization.

The user can define his own event by specifying a number included in the range from 49152 and 50175 with the EVENT condition.

The condition is satisfied when the SIGNAL EVENT statement occurs on the specified event number. For example:

```
CONDITION [90] :
  WHEN EVENT 50100 DO
    $DOUT[25] := OFF
  ENDCONDITION
  <statements...>
  SIGNAL EVENT 50100 -- this triggers condition [90]
```

The programmer can specify a program name or use the reserved word ANY to specify which program the event must be caused by to trigger the user events.

The syntax is as follows:

```
|| By prog_name | BY ANY ||
```

If nothing is specified, the program in which the condition handler is defined is used. For example:

```
WHEN EVENT 49300 DO
WHEN EVENT 49300 BY ANY DO
```

```
WHEN EVENT 49300 BY weld_prog DO
```

8.3.7 ERROR Events

An error event condition monitors the occurrence of an error. The condition is satisfied when the specified error event occurs while the condition handler is enabled. The condition expression is scanned only when an error occurs.

Error events include the following:

- **ANYERROR:** occurrence of any error;
- **ERRORNUM n:** occurrence of an error, identified by number n;
- **ERRORCLASS n:** occurrence of an error belonging to the class identified by one of the following predefined constants:

EC_BYPASS	Cancellation of suspendable statement (52224-52233)
EC_COND	Condition Related Errors (25600-25607)
EC_DISP	Continuous Display Errors (29696-29715)
EC_ELOG	Error Logger Errors (27648-27660)
EC_FILE	File System Errors (768-788)
EC_MATH	Math Conversion Errors (21505-21516)
EC_PIO	File Input/Output Errors (514-583)
EC_PROG	Program Execution Errors (36864-37191)
EC_SYS	System State Errors (28672-28803)
EC_SYS_C	Startup and SYS_CALL Errors (24576-24587)
EC_TRAP	PDL2 Trappable Errors (39937-40108)
EC_USR1	User-Defined Errors (43008-43519)
EC_USR2	User-Defined Errors (43520-44031)

WinC5G Program that runs on PC can be used for getting the documentation related to C5G errors.

The programmer can specify a program name or use the reserved word ANY to specify which program the error must be caused by to trigger the error events.

The syntax is as follows:

```
||BY prog_name | BY ANY ||
```

If nothing is specified, the program in which the condition handler is defined is used.

For example:

```
WHEN ANYERROR DO
WHEN ERRORCLASS EC_FILE BY ANY DO
WHEN ERRORNUM 36939 BY weld_prog DO
```

The value of \$THRD_ERROR for interrupt service routines initiated as actions of error events will be set to the error number that triggered the event.

8.3.8 PROGRAM Events

A program event condition monitors a program generated event. The condition is satisfied when the specified program event occurs while the condition handler is enabled. The condition expression is scanned only when a program event occurs.

Program events include the following:

- **ACTIVATE**: activation of program execution;
- **DEACTIVATE**: deactivation of program execution;
- **PAUSE**: pause of program execution;
- **UNPAUSE**: unpause of paused program;
- **EXIT CYCLE**: exit of current cycle and start of next cycle.

The programmer can specify a program name or use the reserved word ANY to specify any program.

For example:

```
WHEN ACTIVATE DO
WHEN DEACTIVATE weld_prog DO
WHEN PAUSE ANY DO
WHEN UNPAUSE DO
WHEN EXIT CYCLE DO
```

8.3.9 Event on Cancellation of a Suspendable Statement

This class of events can be used for detecting the cancellation of the interpretation of a suspendable statement (MOVE, DELAY, WAIT, WAIT FOR, READ, PULSE, SYS_CALL). This can happen by:

- pressing the ^C key on the current statement in execution when working in a programming environment;
- pressing the arrow up or arrow down key when working in a programming environment;
- issuing the PROGRAM STATE BYPASS command from the system command menu or the BYPASS statement.

The same mechanism used for error events is adopted. There is a new class of errors, included between 52224 and 52233, with the following meaning:

- 52224 - ^C or BYPASS of motion on ARM [1]
- 52225 - ^C or BYPASS of motion on ARM [2]
- 52226 - ^C or BYPASS of motion on ARM [3]
- 52227 - ^C or BYPASS of motion on ARM [4]
- 52228 - ^C or BYPASS of SYS_CALL
- 52229 - ^C or BYPASS of PULSE
- 52230 - ^C or BYPASS of READ
- 52231 - ^C or BYPASS of DELAY
- 52232 - ^C or BYPASS of WAIT
- 52233 - ^C or BYPASS of WAIT FOR

For monitoring the event, the user should write an error event (WHEN ERRORNUM) specifying one of the listed above errors and eventually associate this event, using the BY clause, to the desired program.

The BY clause can be followed by one or more programs or by the reserved word ANY. It is also possible to use the WHEN ERROR CLASS EC_BYPASS condition, for monitoring the entire class of events.

The program associated to the BY clause represents:

- the program that is being executed in a programming environment when the ^C key is pressed meanwhile the cursor is positioned on the suspendable statement;
- the program that was executing the suspendable statement which has been bypassed by a BYPASS Statement or a SYS_CALL of the PROGRAM STATE BYPASS command.

The BY clause can also not be specified if the monitoring of the suspendable statement deletion concerns the program that defines the CONDITION.

Examples:

```

CONDITION [1] :
    WHEN ERRORNUM 52230 BY pippo DO -- ^C or Bypass on READ
executed from pippo
        PAUSE pluto                                -- pause program pluto
    WHEN ERRORNUM 52225 BY ANY DO                 -- ^C or Bypass on MOVE on
ARM 2 executed by any
        PAUSE                                         -- pause this program
ENDCONDITION
ENABLE CONDITION [1]

```

Obviously, it is possible to use this events in a WAIT FOR statement too.

For example:

```
WAIT FOR ERRORNUM 52228 BY pluto
```

8.3.10 MOTION Events

A motion event condition monitors an event related to a motion segment. The condition is satisfied when the specified motion event occurs while the condition handler is enabled. The condition expression is scanned only when a motion event occurs.

Motion events include the following:

- **TIME int_expr AFTER START:** time after the start of motion. **int_expr** represents a time in milliseconds.
- **TIME int_expr BEFORE END:** time before the end of motion. **int_expr** represents a time in milliseconds.
- **DISTANCE real_expr AFTER START:** distance after the start of motion. **real_expr** represents a distance in millimeters.
- **DISTANCE real_expr BEFORE VIA:** distance before VIA position. **real_expr** represents a distance in millimeters.
- **DISTANCE real_expr AFTER VIA:** distance after VIA position. **real_expr** represents a distance in millimeters.
- **DISTANCE real_expr BEFORE END:** distance before the end of motion. **real_expr** represents a distance in millimeters.
- **PERCENT int_expr AFTER START:** % of distance traveled after start of motion;
- **PERCENT int_expr BEFORE END:** % of distance traveled before end of motion;
- **AT START:** start of motion;
- **AT VIA:** VIA position is reached;
- **AT END:** end of motion;

- **RESUME:** motion resumed;
- **STOP:** motion stopped.

The reserved words START and END refer to the start and end of a motion. The reserved word VIA refers to the intermediate position of the motion specified in the VIA clause of the MOVE statement. (VIA is most commonly used for circular motion.) PERCENT refers to the percentage of the distance traveled.

For example:

```
WHEN TIME 200 AFTER START DO
WHEN DISTANCE 500 BEFORE END DO
WHEN PERCENT 30 AFTER START DO
WHEN AT END DO
WHEN AT VIA DO
```

Motion events cannot apply to different arms in the same condition handler. The WHEN PERCENT motion event is only allowed on joint motions.

The condition will never trigger if any of the expression values are outside of the range of the motion segment, for example, if the time before end is greater than the total motion segment time.

Note that, when executing circular motion between PATH nodes, conditions must not be associated to the VIA node as them will not trigger. For this reason the AT VIA condition must be enabled with the destination node of the circular move and not with the via node.

For example:

```
PROGRAM example
TYPE ndef = NODEDEF
  $MAIN_POS
  $MOVE_TYPE
  $COND_MASK
ENDNODEDEF
VAR path_var : PATH OF ndef
  int_var: INTEGER
BEGIN
  CONDITION[5] : -- define the AT VIA condition
    WHEN AT VIA DO
      $FDOUT[5] := ON
  ENDCONDITION
  NODE_APP(path_var, 10) -- append 10 nodes to the PATH path_var
  path_var.COND_TBL[2] := 5 -- fill a condition table element
  FOR int_var := 1 TO 10 DO
    -- assign values to path nodes using the POS built-in function
    path_var.NODE[int_var].$MAIN_POS := POS(...)
    path_var.NODE[int_var].$MOVE_TYPE := LINEAR
    path_var.NODE[int_var].$COND_MASK := 0
  ENDFOR
  -- associate the AT VIA condition to node 5
  path_var.NODE[5].$COND_MASK := 2
  -- define node 4 as the VIA point
  path_var.NODE[4].$MOVE_TYPE := SEG_VIA
  -- define the circular motion between node 3 and 5
```

```

path_var.NODE[5].$MOVE_TYPE := CIRCULAR
CYCLE
    -- execute the move along the path.
    -- The interpolation through nodes will be LINEAR except
    -- between nodes 3.. 5 that will be CIRCULAR.
    MOVE ALONG path_var[1..9]
END example

```

Also refer to [Chap.4. - Motion Control](#) for further detail about conditions in PATH.

STOP, RESUME, AT START, and AT END motion event conditions can be used in condition handlers which are globally enabled. An error is detected if the ENABLE statement or action includes a condition handler containing other motion events. Globally enabled motion events apply to all motion segments. This means they apply to each MOVE statement and to each node segment of a path motion.

Note that STOP, RESUME, and AT END, if used locally in a MOVE statement, will not trigger upon recovery of an interrupted trajectory if the recovery operation is done on the final point.

To detect when the robot stops after a motion statement deactivation (^C or Bypass), it is necessary that the WHEN STOP condition be globally enabled. If it is locally used with the MOVE statement, the condition would not trigger.

When the motion event triggers, the predefined variable \$THRD_PARAM is set to the node number associated to the triggered motion event.

8.4 Actions

The following actions can be included in an *action_list*:

- [ASSIGNMENT Action](#)
- [INCREMENT and DECREMENT Action](#)
- [BUILT-IN Actions](#)
- [SEMAPHORE Action](#)
- [MOTION and ARM Actions](#)
- [ALARM Actions](#)
- [PROGRAM Actions](#)
- [CONDITION HANDLER Actions](#)
- [DETACH Action](#)
- [PULSE Action](#)
- [HOLD Action](#)
- [SIGNAL EVENT Action](#)
- [ROUTINE CALL Action](#)

8.4.1 ASSIGNMENT Action

The assignment action assigns a value to a static user-defined variable, system variable, or output port. The value can be a static user-defined variable, predefined variable, port array item, constant, or literal. A local variable or parameter cannot be used as the value or the assigned variable.

If a variable is used as the right operand of the := operator, it does not have to be

initialized when the condition handler is defined or when the action takes place.

For example:

```
int_var := 4
$DOUT[22] := ON
int_var := other_int_var
```

In the example above, if *other_int_var* is uninitialized when the action takes place, *int_var* will be set to uninitialized.

If an uninitialized boolean variable is assigned to an output port (\$DOUT, \$FDOUT, etc.) the port is automatically set to TRUE. The example below is valid only inside condition actions — an error would be returned in a normal program statement.

```
$DOUT[25] := b -- b is an uninitialized boolean variable
-- $dout[25] will be set to on
```

8.4.2 INCREMENT and DECREMENT Action

The increment action adds a value to an integer program variable. The decrement action subtracts a value to an integer program variable. The value can be a static user-defined variable, constant or literal.

A local variable or parameter cannot be used as the value or as the assigned variable. If a variable is used as the right operand of the *+=* or *-=* operator, it does not have to be initialized when the condition handler is defined or when the action takes place.

For example:

```
int_var += 4
int_var -= 4
int_var += other_int_var
int_var -= other_int_var
```

8.4.3 BUILT-IN Actions

Built-in actions are built-in procedures which can be used as actions of a condition handler. Currently, only the BIT_SET, BIT_CLEAR and BIT_ASSIGN built-in procedures are allowed. When calling a built-in procedure as an action, local variables cannot be used as parameters that are passed by reference. Refer to [Chap.11. - BUILT-IN Routines List](#) for more information on these built-in routines.

For example:

```
BIT_SET(int_var, 1)
BIT_CLEAR($WORD[12], int_var)
BIT_ASSIGN(int_var, 4, bool_var, TRUE, FALSE)
```

8.4.4 SEMAPHORE Action

Semaphore actions clear and signal semaphores. The SIGNAL action releases the specified semaphore. If programs were waiting, the first one is resumed. The CANCEL semaphore action sets the signal counter of the specified semaphore to zero. It is an error if programs are currently waiting on the semaphore. These actions have the same effect as the corresponding statements, as explained in the “Execution Control” chapter.

For example:

```
SIGNAL semaphore_var
CANCEL semaphore_var
```

The *semaphore_var* cannot be a local variable or parameter.

8.4.5 MOTION and ARM Actions

Motion actions cancel or resume motion and arm actions, detach, lock, and unlock arms. These actions have the same effect as their corresponding statements, as explained in [Chap.4. - Motion Control](#). The following actions (excluding SIGNAL SEGMENT) can be used on the default arm, a specified arm or list of arms, or on all arms. The RESUME action will not always produce the order of execution expected in a list of actions. The RESUME action is always executed after all other actions except the routine call action. The rest of the actions are performed in the order in which they occur.

- **CANCEL:** cancels motion
- **SIGNAL SEGMENT:** resumes path motion;
- **DETACH:** detaches an attached arm(s);
- **LOCK:** locks an arm(s);
- **RESUME:** resumes suspended motion which resulted from a LOCK statement;
- **UNLOCK:** unlocks a locked arm(s).

For example:

```
CANCEL CURRENT
CANCEL CURRENT SEGMENT
SIGNAL SEGMENT weld_path
DETACH ARM[1]
LOCK ARM[1], ARM[2]
RESUME ARM[1], ARM[2]
UNLOCK ALL
```

8.4.6 ALARM Actions

The CANCEL ALARM action clears the system alarm state. This action has the same effect as the corresponding statement, as explained in [Chap.10. - Statements List](#).

For example:

```
CANCEL ALARM
```

8.4.7 PROGRAM Actions

Program actions start and stop program execution. These actions have the same effect as their corresponding statements, as explained in [Chap.6. - Execution Control](#).

- **PAUSE:** pauses program execution;
- **UNPAUSE:** unpauses a paused program;
- **DEACTIVATE:** deactivates program execution;
- **ACTIVATE:** activates program execution;
- **EXIT CYCLE:** exits current cycle and starts next cycle;
- **BYPASS:** skips the suspendable statement (READ, WAIT on a semaphore, WAIT FOR, SYS_CALL, DELAY, PULSE, MOVE) currently in execution.

The programmer can specify a program or list of programs. A program must be specified for the ACTIVATE action.

For example:

```
PAUSE
UNPAUSE
DEACTIVATE weld_prog
ACTIVATE util_prog
EXIT CYCLE prog_1, prog_2, prog_3
```

8.4.8 CONDITION HANDLER Actions

Condition handler actions perform the following condition handler operations:

- **ENABLE CONDITION:** enables specified condition handler(s);
- **DISABLE CONDITION:** disables specified condition handler(s);
- **PURGE CONDITION:** purges specified condition handler(s);
- **DETACH CONDITION:** detaches specified condition handler(s).

These actions have the same effect as their corresponding statements, as explained earlier in this chapter. The programmer can specify a single condition or list of conditions.

8.4.9 DETACH Action

The DETACH action is used to detach an attached resource. The resource can be an arm (described above with other arm related actions), a device, a condition handler (described above with other condition handler related actions), or a timer. The DETACH action has the same meaning as the corresponding DETACH statement described in [Chap.10. - Statements List](#).

For example:

```
DETACH $TIMER[2] -- detaches a timer
DETACH CRT2: -- detaches a window device
```

8.4.10 PULSE Action

The PULSE action reverses the current state of a digital output signal for a specified number of milliseconds. This action has the same effect as the corresponding PULSE statement, as explained in [Chap.10. - Statements List](#). The ADVANCE clause must be used when PULSE is used as an action.

For example:

```
PULSE $DOUT[21] FOR 200 ADVANCE
```

8.4.11 HOLD Action

The HOLD action causes the system to enter the HOLD state. This means all running holdable programs are placed in a ready state and all motion stops.

8.4.12 SIGNAL EVENT Action

The SIGNAL EVENT action causes the corresponding user event, included in the range 49152-50175, being triggered, if any is defined and enabled in the system. This action has the same effect as the corresponding statement, as explained in [Chap.10. - Statements List](#).

For example:

```
SIGNAL EVENT 50000
```

8.4.13 ROUTINE CALL Action

A routine call action interrupts program execution to call the specified procedure routine. The following restrictions apply to interrupt routines:

- the routine must be user-defined or one of the special built-in routines allowed by the built-in action;
- the routine cannot have more than 16 parameters and must be a procedure;
- all arguments must be passed by reference except INTEGER, REAL, and BOOLEAN arguments;
- all arguments passed by reference must have a main program context scope (not local to a routine);
- the calling program is suspended while the interrupt routine is executed;
- interrupt routines can be interrupted by other interrupt routines.

Arguments passed by value to an interrupt routine use the value at the time the condition handler is defined (not when it triggers).

Each interrupt routine gets a new copy of the \$THRD_CEXP, \$THRD_ERROR and \$THRD_PARAM predefined variable. These are initialized to 0 unless the interrupt routine is activated. If the interrupt is caused by an error event, \$THRD_ERROR is initialized to the error number causing the event to trigger.

Interrupt service routines use the same arm, priority, stack, program-specific predefined variable values, and trapped errors as the program which they are interrupting.

Interrupt service routines should be written as short as possible and should not use suspendable statements (WAIT FOR, DELAY, etc). Interrupt service routines are executed by the interpreter and are multi-tasked with all other programs running in the system. If another running program has a higher priority than the program being interrupted the routine will not execute until the higher priority program is suspended. An interrupt service routine can also be interrupted by other condition handlers being activated. Interrupt service routines that are suspendable or require a long time to execute can cause the program stack to become full or drain the system of vital resources.

A way to disallow the interruption of execution by another interrupt service routine is given by the DISABLE INTERRUPT statement. Refer to [Chap.10. - Statements List](#).

8.4.14 Execution Order

Actions are performed in the order in which they are listed. PDL2 syntax requires that all actions that are interrupt routines be listed last, meaning they are executed last. The only exception to this rule is the RESUME action. In the following example the order of execution for the actions is not what would be expected

```
CONDITION[1] :
WHEN $DIN[1] DO
    UNLOCK ARM[1] -- Unlock arm to allow restart of motion
    RESUME ARM[1] -- Resume motion on the arm
    $DOUT[1] := ON -- Signal the outside world we started motion
    user_isr -- User defined routine
ENDCONDITION
```

The shown above example, if executed outside of a condition handler, would produce the results expected by unlocking the arm, resuming the motion, setting the signal, and then calling the routine. When such a code segment is executed inside the condition statement, the digital output will be set before the RESUME action is executed. This is because the RESUME action is always executed after all other actions except the routine call.

9. COMMUNICATION

This chapter explains the PDL2 facilities for handling stream input and output. Stream I/O includes reading input from and writing output to operator devices, data files, and communication ports. Depending on the device, the input and output can be in the form of ASCII text or binary data.

A tool, called WinC5G, is provided by COMAU to be run on a PC (for further information see Chapter **WinC5G Program** in the **C5G Control Unit Use Manual**). One of the features of such a Program is the possibility of opening a Terminal that allows to issue commands and view information about the Robot Controller Unit.

The e-mail functionality is described in this chapter too, as an important way of communicating.

- [Devices](#)
- [Logical Unit Numbers](#)
- [E-Mail functionality.](#)

9.1 Devices

PDL2 supports the following types of devices:

- [WINDOW Devices](#)
- [FILE Devices](#)
- [PIPE Devices](#)
- external;
- [PDL2 Devices](#)
- [NULL Device](#)
- Serial line
- Network (UDP and TCP)

Devices are specified in a PDL2 program as a string value. Each device name consists of one to four characters followed by a colon. For example, 'CRT:' which specifies the scrolling window on the system screen of the Teach Pendant.

9.1.1 WINDOW Devices

Window devices are areas on the Teach Pendant (scrolling area of TP-INT Page) and WinC5G (Terminal Window) screens to which a program can write data and from which a program can read data that is entered with the PC keyboard or Teach Pendant keypad. Window devices use ASCII format. Windows allow a program to prompt the operator to take specific actions or enter information and to read operator requests or responses.

PDL2 recognizes the following window device names:

CRT(WinC5G Terminal Window) Teach Pendant (scroll area in TP-INT Page)

CRT: (default)

TP: (default)

CRT1:

TP0:

CRT(WinC5G Terminal Window) Teach Pendant (scroll area in TP-INT Page)

CRT2 : TP1 :

CRT3 : TP2 :

TP3 :

CRT: and TP: indicate the scrolling window on the system screen. TP0: indicates the Teach Pendant character menu window which can be popped up on windows of the user screen. (Refer to [BUILT-IN Routines List](#) chapter for a description of the window related predefined routines.) The other windows, CRT1-3 and TP1-3, indicate windows on WinC5G and Teach Pendant user screen. The user screen is divided into three areas, numbered 1 through 3 from top to bottom.

The Screen key switches between the system and user screens. Screen built-in routines can be used to determine which screen is currently displayed and force a particular screen to be displayed (refer to [BUILT-IN Routines List](#) chapter).

PDL2 provides built-in routines to perform window operations, including creating and displaying user-defined windows, positioning the cursor, clearing windows, and setting window attributes (refer to [BUILT-IN Routines List](#) chapter). The first element of the predefined array variable \$DFT_DV indicates the default window used in the window related built-in routines.

When the selected window is 'TP:' on the system screen, and there is a read active on that window, the user is sent the message "Input directed to other window. Press SEL key." This indicates that the input is directed to a window other than the command menu. The user must press the SEL key to get back to the command menu.

WINDOW SELECT condition can be used in a condition handler or WAIT FOR Statement to determine when a specific window has been selected for input. Refer to [Chap.8. - Condition Handlers](#) for more information on conditions and condition handlers.

Some tests can be performed on the CRT device since it is a disconnectable device, and the CRT could be absent when accessing it. A problem can develop when the CRT connection is disconnected when a read is pending on it. To prevent this, the PDL2 programmer can determine the presence of the CRT emulator by testing the value of the \$SYS_STATE predefined variable. It is also possible to define a condition handler to trigger when the CRT emulator protocol is connected and disconnected. Please refer to [Chap.8. - Condition Handlers](#) for more information on using condition handlers in this way.

9.1.2 FILE Devices

File devices are secondary storage devices. Programs can read data from or write data to files stored on these devices. A file is a collection of related information that is stored as a unit. File devices allow a program to store and retrieve data in ASCII or binary format.

PDL2 recognizes the following file device names:

- UD: (User disk)
- TD: (Temporary Device)
- TX: (Teach Pendant External Device)
- XD: (External Device)

- UD: is a User Disk. This is the main storage device. The user is allowed to store herein program and data files, organized in directories if needed.

- TD: is a Temporary device which can be used for temporary storing files. Upon power failure this device is cleared and inside stored data are lost.
- TX: is an external device which is recognized by the system when the USB flash disk is inserted in the USB port of the Teach Pendant
- XDn: are eXternal devices which are recognized by the system when the USB flash disk is inserted in the USB ports. There are 5 available eXternal devices: XD:, XD2:, XD3:, XD4:, XD5:

The TD: file device refers to the Controller RAM disk.

PDL2 provides built-in routines to perform file operations such as positioning a file pointer or determining the number of remaining bytes in a file (refer to [Chap.11. - BUILT-IN Routines List](#)).

9.1.3 PIPE Devices

The PIPE devices provide a mechanism that lets programs communicate with each other through the standard Serial I/O interface.

Programs can read data from or write data to PIPEs with normal [READ Statements](#) and [WRITE Statements](#).

A PIPE is created using the DV_CNTRL(1) and can be deleted with DV_CNTRL (2). For further details see [par. 11.43 DV_CNTRL Built-In Procedure on page 256](#).

It can be opened using the [OPEN FILE Statement](#) providing the device identifier as pipe and then the name of the pipe. eg. 'pipe:example'.

When creating a pipe it is possible to specify that the pipe is deleted when no more LUNs are opened on it or when the program which created it is deactivated. This is useful for automatic cleanup of resources.

The format of the data in the pipe is not defined by the system but is up to the user. What is written - is read.

9.1.4 PDL2 Devices

PDL2 devices are ports through which PDL2 programs can communicate with other devices connected to the Controller.

PDL2 recognizes the following device names:

COM0 :	COM1 :	COMP :
DGM1 :	DGM2 :	DGM3 :
NET1 :	NET2 :	NETP :
MDM :		

COMP: is the COMAU channel for WinC5G Program interfacing.

COM0: and COM1: are COMAU serial ports that are available for the user.

DGM1:..DGM3: these are other names for COMAU COM0: / COM1:, when the 3964R protocol has been mounted on it.

NETn: are the channels of communication for various local area networking configurations:

NET0: is the default device upon which WinC5G communicates when mounted with TCP/IP

NET1: and NET2: are FTP client devices

NETP: is the channel when PPP protocol is used

NETT: is the channel when TCP protocol is used

NETU: is the channel when UDP protocol is used

USBC: is the indicator of the USB port

MDM: is the mounted modem device

PDL2 devices can have communication protocols mounted on them. If 3964R is mounted on a device, it is referenced using the appropriate DGMn: name. Otherwise, it is referenced using the corresponding COMn: name. Data can be transmitted in ASCII or binary format by first opening a LUN on the device and then reading or writing the data. Refer to later sections of this chapter for information on opening a LUN and transmitting data.

When the modem is mounted on a port (UCMM), then it is named device MDM:. Such a name can be used for accepting/connecting over the modem and also for streamed input/output.

[DV_CNTRL Built-In Procedure](#) can be used for getting and setting the PDL2 device characteristics.

These are identified by the following predefined constants:

```

COM_BD110    : 100    baud transmission rate
COM_BD300    : 300    baud transmission rate
COM_BD1200   : 1200   baud transmission rate
COM_BD2400   : 2400   baud transmission rate
COM_BD4800   : 4800   baud transmission rate
COM_BD9600   : 9600   baud transmission rate
COM_BD19200  : 19200  baud transmission rate
COM_BD38400  : 38400  baud transmission rate
COM_BD57600  : 57600  baud transmission rate
COM_BD115200 : 115200 baud transmission rate

COM_PAR_ODD  : odd parity
COM_PAR_EVEN : even parity
COM_PAR_NO   : no parity

COM_STOP1    : 1      stop bit
COM_STOP2    : 2      stop bits
COM_STOP1_5  : 1.5   stop bits

COM_BIT7     : 7 bits per character are transferred
COM_BIT8     : 8 bits per character are transferred

COM_XSYNC    : XON/XOFF flow control used
COM_XSYNC_NO : no XON/XOFF flow control

COM_RDAHD    : use a large (384 bytes)readahead buffer that
guarantees faster Read operations and no characters are lost
COM_RDAHD_NO : no readahead buffer used

COM_CHAR     : use 7-bit ASCII code (only characters between 32
and 126 of the ASCII table are read through the communication
port).
COM_PASAL    : all received characters are passed through the
communication port.

```

COM_CHARNO : of each received character, the 7th bit is cleared and, if the resulting value stays in the range 32-126, the whole character, 8th bit included, is read.

COM_CHAR and **COM_PASAL** are not mutually exclusive; if they are put in OR together, the characters read are included in the range 0-127 of ASCII characters table.

The following table gives a clearer representation of the effect of these attributes ("yes" means that the character is read; "no" that the character is ignored):

Tab. 9.1 - Received characters through a communication port

Received Characters	COM_CHARNO	COM_CHAR	COM_PASAL	COM_CHAR OR COM_PASAL
00-31, 127	no	no	yes	yes
32-126	yes	yes	yes	yes
128-159, 255	no	no	yes	no
160-254	yes	no	yes	no

9.1.4.1 Sample Program for PDL2 Serial device use

```

PROGRAM tcom NOHOLD

VAR vs_data : STRING[80] NOSAVE
VAR vi_lun_r : INTEGER NOSAVE
VAR vi_lun_w : INTEGER NOSAVE

BEGIN
    ERR_TRAP_ON(39990) -- "Handle READ/Filer error on PDL2 program"
    $ERROR := 0
    DV_CNTRL(4, 'COM0:', COM_BD115200 OR COM_RDAHD OR COM_BIT8 OR COM_PAR_NO OR
    COM_STOP1 OR COM_PASAL, 65536)
    IF $ERROR <> 0 THEN
        WRITE LUN_CRT ('Error opening Serial Channel ', $ERROR, NL)
        PAUSE
    ENDIF
    OPEN FILE vi_lun_r ('COM2:', 'r') WITH $FL_BINARY = ON,
    OPEN FILE vi_lun_w ('COM2:', 'w') WITH $FL_BINARY = ON,
    CYCLE
    READ vi_lun_r (vs_data::8)
    WRITE LUN_CRT ('Received =', vs_data, NL)

    WRITE vi_lun_w (vs_data) -- invio echo
    WRITE LUN_CRT ('Sent =', vs_data, NL)
END tcom

```

9.1.5 NULL Device

The null device is used to redirect output to a non-existent device. It is typically used in error handling, or as an aid in debugging programs that are not yet complete. Data that is written to the null device is thrown away. The null device can use ASCII or binary format. The pre-defined constant LUN_NULL may be used to reference the null device. PDL2 recognizes the string 'NULL:' as the null device.

9.2 Logical Unit Numbers

A logical unit number (LUN) represents a connection between a program and a physical device which the program can communicate with.

The following predefined LUNs are recognized as already being opened for I/O operations:

LUN	Devices
LUN_TP	TP :
LUN_CRT	CRT :
LUN_NULL	NULL :

The predefined variable \$DFT_LUN indicates the default LUN for I/O operations.

The default LUN is the Teach Pendant, and remains the Teach Pendant even if the Terminal of WinC5G is active. CRT emulator protocol is mounted on a port. If the PDL2 programmer wants to redirect the output to the CRT: device on the Terminal, the \$DFT_LUN predefined variable can be set to LUN_CRT value.



For further information about using LUNs, see [Chap.10. - Statements List](#):
[CLOSE FILE Statement](#),
[OPEN FILE Statement](#),
[READ Statement](#),
[WRITE Statement](#).

9.3 E-Mail functionality

Current section describes the e-mail functionality which allows to send/receive e-mails by means of the C5G Controller Unit. The currently supported protocols are SMTP and POP3.

To use the e-mail capability, the following parameters are needed:

- **name / IP address** of the SMTP server
- **name / IP address** of the POP3 server
- **login** and **password** of the POP3 server

A detailed description is supplied about the following topics:

- [Configuration of SMTP and POP3 clients](#)
- [Sending/receiving e-mails on C5G Controller](#)
- [Sending PDL2 commands via e-mail](#)

9.3.1 Configuration of SMTP and POP3 clients

Two predefined variables are available to configure SMTP and POP3 clients:

- [\\$EMAIL_STR: Email string configuration](#)
- [\\$EMAIL_INT: Email integer configuration](#)

Their fields have the following meaning:

Tab. 9.2 - \$EMAIL_STR

[1]: POP3 server address or name for the incoming email
[2]: SMTP server address or name for the outgoing email
[3]: sender e-mail address
[4]: login of the POP3 server
[5]: password of the POP3 server
[6]: directory where attachments are saved. It will be a subdirectory of UD:\SYS\EMAIL

Tab. 9.3 - \$EMAIL_INT

[1]: flags
<ul style="list-style-type: none"> • 0x01: Feature is enabled • 0x04: use APOP authentication • 0x10: Use of memory or file system for incoming mail • 0x020: Do not remove the directory where attachments are stored upon system startup
[2]: Maximum size in bytes for incoming messages (if 0, the default is 300K, that is 300 * 1024 bytes)
[3]: Polling interval for POP3 server (if 0, the default and minimum used value is 60000 ms)
[4]: Size to reserve to the body of the email (if 0 the default value is 5K, that is 5 * 1024 bytes)
[5]: Maximum execution time for command (if 0 the default value is 10000 ms)

Both arrays are retentive, thus it is needed to issue a ConfigureControllerSave command to save them.

9.3.2 Sending/receiving e-mails on C5G Controller

A PDL2 program called “email” is shown below (“email” program): it allows to send and receive e-mails on C5G Controller.

[DV_CNTRL Built-In Procedure](#) is to be used to handle such functionalities.



See [DV_CNTRL Built-In Procedure](#) in [Chap. BUILT-IN Routines List](#) section for further information about the e-mail functionality parameters.

9.3.2.1 “email” program

```

PROGRAM email NOHOLD, STACK = 10000
CONST ki_email_cfg = 20
  ki_email_send = 21
  ki_email_num = 22
  ki_email_recv = 23
  ki_email_del = 24
  ki_email_hdr = 25
  ki_email_close = 26
VAR ws_cfg : ARRAY[6] OF STRING[63]
  wi_cfg : ARRAY[5] OF INTEGER
  si_handle : INTEGER
  vs_insrv, vs_outsrv : STRING[63]
  vs_replyto, vs_login, vs_password : STRING[63]
  vs_inbox : STRING[255]
  vs_from : STRING[63]
  vs_subj, vs_body : STRING[1023]
  ws_attachments : ARRAY[10] OF STRING[64]
  vi_num, vi_date, vi_ans, vi_ext, vi_first : INTEGER
  vs_input : STRING[1]
  ws_to_cc : ARRAY[2,8] OF STRING[63]

ROUTINE pu_help
BEGIN
  WRITE LUN_CRT ('1. Get Number of mail on the server', NL)
  WRITE LUN_CRT ('2. Send a new mail', NL, '3. Receive a mail', NL, '4. Get
header only', NL)
  WRITE LUN_CRT ('5. Delete a mail from server', NL, '6. This
menu', NL, '7. Quit', NL)
END pu_help

ROUTINE pu_getmenu(as_input : STRING) : INTEGER
BEGIN
  IF as_input = '0' THEN
    RETURN(0)
  ENDIF
  IF as_input = '1' THEN
    RETURN(1)
  ENDIF
  IF as_input = '2' THEN
    RETURN(2)
  ENDIF
  IF as_input = '3' THEN
    RETURN(3)
  ENDIF
  IF as_input = '4' THEN
    RETURN(4)
  ENDIF
  IF as_input = '5' THEN
    RETURN(5)
  ENDIF
  IF as_input = '6' THEN
    RETURN(6)
  ENDIF
  IF as_input = '7' THEN
    RETURN(7)
  ENDIF

```

```

ENDIF
IF as_input = '8' THEN
    RETURN(8)
ENDIF
IF as_input = '9' THEN
    RETURN(9)
ENDIF
RETURN(-1)

END pu_getmenu

ROUTINE pu_getstrarray(as_prompt : STRING; aw_input : ARRAY[*] OF STRING; ai_n :
INTEGER)
VAR vi_i : INTEGER
VAR vi_skip : INTEGER
BEGIN
    vi_i := 1
    vi_skip := 0
    WHILE vi_i <= ai_n DO
        IF vi_skip = 0 THEN
            WRITE LUN_CRT (as_prompt, '(', vi_i, ')', ': ')
            READ LUN_CRT (aw_input[vi_i])
            IF aw_input[vi_i] = '' THEN
                vi_skip := 1
            ENDIF
        ELSE
            aw_input[vi_i] := ''
        ENDIF
        vi_i += 1
    ENDWHILE
END pu_getstrarray

ROUTINE pu_getstrmatrix2(as_prompt1 : STRING; as_prompt2 : STRING; aw_input :
ARRAY[*,*] OF STRING; ai_n : INTEGER)
VAR vi_i : INTEGER
VAR vi_skip : INTEGER
BEGIN
    vi_i := 1
    vi_skip := 0
    WHILE vi_i <= ai_n DO
        IF vi_skip = 0 THEN
            WRITE LUN_CRT (as_prompt1, '(', vi_i, ')', ': ')
            READ LUN_CRT (aw_input[1,vi_i])
            IF aw_input[1,vi_i] = '' THEN
                vi_skip := 1
            ENDIF
        ELSE
            aw_input[1,vi_i] := ''
        ENDIF
        vi_i += 1
    ENDWHILE
    vi_i := 1
    vi_skip := 0
    WHILE vi_i <= ai_n DO
        IF vi_skip = 0 THEN
            WRITE LUN_CRT (as_prompt2, '(', vi_i, ')', ': ')
            READ LUN_CRT (aw_input[2,vi_i])
        ENDIF
        vi_i += 1
    ENDWHILE

```

```

IF aw_input[2,vi_i] = '' THEN
    vi_skip := 1
ENDIF
ELSE
    aw_input[2,vi_i] := ''
ENDIF
vi_i += 1
ENDWHILE
END pu_getstrmatrix2

ROUTINE pu_getstr(as_prompt, as_input : STRING)
VAR vi_i : INTEGER
BEGIN
    WRITE LUN_CRT (as_prompt, ': ')
    READ LUN_CRT (as_input)
END pu_getstr

BEGIN
    ERR_TRAP_ON(40024)
    ERR_TRAP_ON(39990)
    ERR_TRAP_ON(39995)
    vi_ext := 0
    vi_first := 1
    DELAY 1000

    WRITE LUN_CRT ('TEST 2', NL)

    ws_cfg[1] := '192.168.9.244'
    ws_cfg[2] := '192.168.9.244'
    ws_cfg[3] := 'username@domainname.com'
    ws_cfg[4] := 'proto1'
    ws_cfg[5] := 'proto1'
    ws_cfg[6] := 'pdl2'
    wi_cfg[1] := 0
    wi_cfg[2] := 0
    wi_cfg[3] := 0
    wi_cfg[4] := 0
    wi_cfg[5] := 0
    si_handle := 0

    DV_CNTRL(ki_email_cnfg, si_handle, ws_cfg, wi_cfg)

    WHILE vi_ext = 0 DO
        IF vi_first = 1 THEN
            vi_first := 0
            pu_help
        ENDIF
        WRITE LUN_CRT (NL, 'Command > ')
        READ LUN_CRT (vs_input)
        vi_ans := pu_getmenu(vs_input)
        IF NOT VAR_UNINIT(vi_ans) THEN
            SELECT vi_ans OF
            CASE (1): -- Number of emails
                DV_CNTRL(ki_email_num, (si_handle), vi_num)
                WRITE LUN_CRT (vi_num, ' E-mail waiting on server.', NL)
                WRITE LUN_CRT ('*Mail num OK.', NL)
        ENDIF
    ENDWHILE

```

```

CASE (2): -- Send
    pu_getstrmatrix2('To ', 'Cc ', ws_to_cc, 8)
    pu_getstr ('Subj', vs_subj)
    pu_getstrarray('Att ', ws_attachments, 10)
    pu_getstr ('Body', vs_body)
    DV_CNTRL(ki_email_send, (si_handle), ws_to_cc, (vs_subj), (vs_body),
ws_attachments, 1)
    WRITE LUN_CRT ('*Mail send OK.', NL)
CASE (3): -- Recv
    DV_CNTRL(ki_email_recv, (si_handle), 1, vs_from, vs_subj, vs_body,
vi_date, ws_to_cc, ws_attachments)
    WRITE LUN_CRT ('Received mail:', NL)
    WRITE LUN_CRT (' Date:', vi_date, NL)
    WRITE LUN_CRT (' From:', vs_from, NL)
    WRITE LUN_CRT (' To: ', ws_to_cc[1,1], ' ', ws_to_cc[1,2], ' ',
ws_to_cc[1,3], ' ', ws_to_cc[1,4], ' ', ws_to_cc[1,5], ' ', ws_to_cc[1,6], ' ',
ws_to_cc[1,7], ' ', ws_to_cc[1,8], NL)
    WRITE LUN_CRT (' Cc: ', ws_to_cc[2,1], ' ', ws_to_cc[2,2], ' ',
ws_to_cc[2,3], ' ', ws_to_cc[2,4], ' ', ws_to_cc[2,5], ' ', ws_to_cc[2,6], ' ',
ws_to_cc[2,7], ' ', ws_to_cc[2,8], NL)
    WRITE LUN_CRT (' Subj: ', vs_subj, NL)
    WRITE LUN_CRT (' Att.: ', ws_attachments[1], ' ', ws_attachments[2], ' ',
ws_attachments[3], ' ', ws_attachments[4], ' ', ws_attachments[5], ' ',
ws_attachments[6], ' ', ws_attachments[7], ' ', ws_attachments[8], ' ',
ws_attachments[9], ' ', ws_attachments[10], NL)
    WRITE LUN_CRT (' Body: ', NL, vs_body, NL)
    WRITE LUN_CRT ('*Mail recv OK.', NL)
CASE (4): -- Header
    DV_CNTRL(ki_email_hdr, (si_handle), 1, vs_from, vs_subj, vi_date,
ws_to_cc)
    WRITE LUN_CRT ('Header:', NL)
    WRITE LUN_CRT (' Date:', vi_date, NL)
    WRITE LUN_CRT (' From:', vs_from, NL)
    WRITE LUN_CRT (' To: ', ws_to_cc[1,1], ' ', ws_to_cc[1,2], ' ',
ws_to_cc[1,3], ' ', ws_to_cc[1,4], ' ', ws_to_cc[1,5], ' ', ws_to_cc[1,6], ' ',
ws_to_cc[1,7], ' ', ws_to_cc[1,8], NL)
    WRITE LUN_CRT (' Cc: ', ws_to_cc[2,1], ' ', ws_to_cc[2,2], ' ',
ws_to_cc[2,3], ' ', ws_to_cc[2,4], ' ', ws_to_cc[2,5], ' ', ws_to_cc[2,6], ' ',
ws_to_cc[2,7], ' ', ws_to_cc[2,8], NL)
    WRITE LUN_CRT (' Subj: ', vs_subj, NL)
CASE (5): -- Delete
    DV_CNTRL(ki_email_del, (si_handle), 1)
    WRITE LUN_CRT ('*Delete OK.', NL)
CASE (6): -- Help
    pu_help
CASE (7): -- Exit
    DV_CNTRL(ki_email_close, si_handle)
    WRITE LUN_CRT ('*Program exit.', NL)
    vi_ext := 1
ELSE:
    pu_help
ENDSELECT
ELSE
ENDIF
ENDWHILE
END email

```

9.3.3 Sending PDL2 commands via e-mail

The user is allowed to send PDL2 commands to the C5G Controller Unit, via e-mail. To do that, the required command is to be inserted in the e-mail title with the prefix 'CL' and the same syntax of the strings specified in SYS_CALL built-in. Example: if the required command is ConfigureControllerRestartCold, the user must insert the following string in the e-mail title: 'CL CCRC'.

The authentication is performed by inserting a text which is automatically generated by *C5Gmp* program (on a PC), in the message body. Such a program asks the user the system identifier (\$BOARD_DATA[1].SYS_ID), the sender of the e-mail which includes the required command, the user login and password; it gives back a message to be inserted into the message body, and it will work as an authentication. Note that the PC time and the Controller time (as well as the corresponding timezones) must be synchronized, because the message returned by *C5Gmp* program is ok within a time interval of half an hour, more or less, since the generation time.

10. STATEMENTS LIST

10.1 Introduction

This chapter is an alphabetical reference of PDL2 statements. The following information is provided for each statement:

- short description;
- syntax;
- comments concerning usage;
- example;
- list of related statements.

This chapter uses the syntax notation explained in the "Introduction to PDL2" chapter to represent PDL2 statements.

The available PDL2 statements are:

- ACTIVATE Statement
- ATTACH Statement
- BEGIN Statement
- BYPASS Statement
- CALL Statement
- CALLS Statement
- CANCEL Statement
- CLOSE FILE Statement
- CLOSE HAND Statement
- CONDITION Statement
- CYCLE Statement
- CONST Statement
- DEACTIVATE Statement
- DECODE Statement
- DELAY Statement
- DETACH Statement
- DISABLE CONDITION Statement
- DISABLE INTERRUPT Statement
- ENABLE CONDITION Statement
- ENCODE Statement
- END Statement
- EXIT CYCLE Statement
- FOR Statement

- GOTO Statement
- HOLD Statement
- IF Statement
- IMPORT Statement
- LOCK Statement
- MOVE Statement
- MOVE ALONG Statement
- OPEN FILE Statement
- OPEN HAND Statement
- PAUSE Statement
- PROGRAM Statement
- PULSE Statement
- PURGE CONDITION Statement
- READ Statement
- RELAX HAND Statement
- REPEAT Statement
- RESUME Statement
- RETURN Statement
- ROUTINE Statement
- SELECT Statement
- SIGNAL Statement
- TYPE Statement
- UNLOCK Statement
- UNPAUSE Statement
- VAR Statement
- WAIT Statement
- WAIT FOR Statement
- WHILE Statement
- WRITE Statement

10.2 ACTIVATE Statement

The ACTIVATE statement activates a loaded program. The effect of activating a program depends on the holdable/non-holdable program attribute.

Syntax:

```
ACTIVATE prog_name <, prog_name>...
```

Comments:

prog_name is an identifier indicating the program to be activated. A list of programs can be specified.

If the statement is issued from a non-holdable program, holdable programs are placed in the ready state and non-holdable programs are placed in the running

state. If the statement is issued from a holdable program, the programs are placed in the running state.

The programs must be loaded in memory or an error results. Only one activation of a given program is allowed at a given time.

When a program is activated, the following occurs for that program:

- initialized variables are set to their initial values.
- if *prog_name* is holdable and does not have the DETACH attribute, the arm will be attached. If *prog_name* is non-holdable and has the ATTACH attribute then the arm will be attached.

The ACTIVATE statement is permitted as a condition handler action. Refer to the [Condition Handlers](#) chapter for further information.

Examples:

```
ACTIVATE weld_prog, weld_util, weld_cntrl
```

See also:

[DEACTIVATE Statement](#)

10.3 ATTACH Statement

The ATTACH statement allows a program to gain exclusive control of a resource so that other programs cannot access it. This statement applies to arms, I/O devices, condition handlers, and timers.

Syntax:

```
ATTACH || ARM <ALL> | ARM[n] <, ARM[n]>... ||  
ATTACH device_str <, device_str>...  
ATTACH CONDITION[n] <, CONDITION[n]>...  
ATTACH $TIMER[n] <, $TIMER[n]>...
```

Comments:

When an arm is attached, it is attached to the program executing the ATTACH statement. Other programs cannot cause motion on that arm. The programmer can specify an arm, a list of arms, or all arms. If nothing is specified, the default arm is attached.

It is only a warning if a program attempts to attach an arm that it already has attached. An error occurs if a program attempts to attach an arm that is currently moving or to which another program is already attached.

An error occurs if a program attempts to execute a MOVE or RESUME statement of an arm that is currently attached to another program.

When a device is attached, it is attached to the program executing the ATTACH statement.

The *device_str* can be any I/O device name including the following predefined window and communication devices:

Communication Devices:	Window Devices:
COM1:	CRT:
COM2:	CRT1:
COM3:	CRT2:
COM4:	CRT3:
COMP:	TP:
	TP0:
	TP1:
	TP2:
	TP3:

Communication Devices:	Window Devices:
-------------------------------	------------------------

DGM0:	
-------	--

DGM1:	
-------	--

DGM2:	
-------	--

DGM3:	
-------	--

DGM4:	
-------	--

When an I/O device is attached, other programs cannot open a LUN on that device. An error occurs if a program attempts to attach a device on which LUNs are already opened or to which another program is already attached.

In addition, if the attached device is a window, other programs cannot use that window in the window related built-in routines.

When a condition handler is attached, only code belonging to the same program containing the ATTACH can enable, disable, purge, detach, or redefine the condition handler.

An error occurs if a program attempts to attach a condition handler that does not exist.

When a timer is attached, it is attached to the program owning the ATTACH statement. Only code belonging to the same program containing the ATTACH statement can access the timer. The value of the timer is not changed when it is attached.

To release an arm, device, condition handler, or timer for use by other programs, the DETACH statement must be issued from the same program that issued the ATTACH. This means the DETACH statement must be executed by the attaching program for arms and devices and the DETACH statement must be owned by the attaching program for condition handlers and timers.

When a program terminates or is deactivated, all attached resources are detached. The following example and table illustrate the program considered to have the attached resource:

```

PROGRAM p1                      PROGRAM p2
ROUTINE r2 EXPORTED FROM p2    ROUTINE r2 EXPORTED FROM p2
ROUTINE r1                      ROUTINE r2
BEGIN                           BEGIN
      r2                         -- ATTACH statement
END r1                          END r2
BEGIN                           BEGIN
      r1                         END p2
END p1

```

Examples:

```
ATTACH 'COM1:' -- attach a communication device
```

```
ATTACH 'CRT2:' -- attach a window device
```

```
ATTACH ARM -- attach the default arm
ATTACH ARM[1], ARM[3]
```

```
ATTACH CONDITION[5], CONDITION[8] -- attach condition
handlers
```

```
ATTACH $TIMER[6] -- attach a timer
```

See also:

[DETACH Statement](#)
[PROGRAM Statement \(ATTACH attributes\)](#)
[PROGRAM Statement \(DETACH attributes\)](#)

10.4 BEGIN Statement

The BEGIN statement marks the start of executable code in a program or routine.

Syntax:

```
BEGIN <CYCLE>
```

Comments:

The constant, type, and variable declaration sections must be placed before the BEGIN statement.

The CYCLE option is allowed for any program BEGIN, but cannot be used for a routine BEGIN.

The CYCLE option creates a continuous cycle. When the program END statement is encountered, execution continues back at the BEGIN statement. The cycle continues until the program is deactivated or an EXIT CYCLE statement is executed. An EXIT CYCLE statement causes the termination of the current cycle. Execution immediately continues back at the BEGIN statement.

The CYCLE option is not allowed on the BEGIN statement if the program contains a CYCLE statement.

Examples:

<pre>PROGRAM example_1 -- declaration section BEGIN -- executable section END example1 PROGRAM example3 -- declaration section BEGIN -- initial executable section CYCLE -- cycled executable section END example3</pre>	<pre>PROGRAM example2 -- declaration section BEGIN CYCLE -- executable section END example2</pre>
--	---

See also:

[CYCLE Statement](#)
[DEACTIVATE Statement](#)
[END Statement](#)
[EXIT CYCLE Statement](#)

10.5 BYPASS Statement

The BYPASS statement is used to bypass the current statement in execution of a program, if that statement is a suspendable one. By suspendable statement it is meant

to be one of the following: READ, WAIT FOR, MOVE, SYS_CALL, DELAY, WAIT, PULSE.

Syntax:

```
BYPASS <flags> <| |> prog_name <, <prog_name>... | ALL ||>
```

Comments:

flags is an integer mask used for indicating which kind of suspendable statement should be bypassed. Possible mask values are:

0:	for bypassing the current statement (if suspendable)
64:	for bypassing a READ
128:	for bypassing a WAIT FOR
256:	for bypassing a MOVE
512:	for bypassing a SYS_CALL
1024:	for bypassing a DELAY
2048:	for bypassing a WAIT on a semaphore
4096:	for bypassing a PULSE

These integer values are the same as those returned by the PROG_STATE built-in function.

After having bypassed a MOVE statement, the START button need to be pressed for continuing the execution of the bypassed program.

If *prog_name* or a list of names is specified, those programs are paused. If no name is included, the suspendable statement of the program issuing the BYPASS is bypassed.

In this case, the BYPASS should be issued from a condition handler action when the main of the program is stopped on a suspendable statement.

The BYPASS statement is permitted as a condition handler action. Refer to the [Condition Handlers](#) chapter for further details.

Examples:

```
BYPASS 0 ALL -- bypassed all programs that are executing a
-- suspendable statement
```

```
state:=PROG_STATE (pippo)
IF state = 2048 THEN
  BYPASS state pippo -- bypasses pippo if it is waiting on a
semaphore
ENDIF
```

See also:

[PROG_STATE Built-In Function](#)
[Condition Handlers](#) chapter

10.6 CALL Statement

The CALL statement allows the MAIN section of a program to be called (executed) from within another program. As soon as the execution of the called program is terminated, the control returns back to the calling program and continues its execution from the next statement (after the CALL statement). No arguments can be passed to the MAIN

sections.

Syntax:

```
CALL prog_name
```

Comments:

prog_name is an identifier indicating the program whose MAIN is being called.

Example:

```
CALL prg_1 -- Execution jumps to the BEGIN of the MAIN section
      -- of prg_1 program
```



Upon CYCLE Statement in the called program, the \$CYCLE predefined variable is not incremented, which means that the statement is not executed.

10.7 CALLS Statement

The CALLS statement allows to symbolically call a routine (with or without passing arguments) or the MAIN of a specified program. The program name and the routine name are passed as STRINGS.

Syntax:

```
CALLS (prog_name <, rout_name>) <(arg_val <, arg_val>...)>
```

Comments:

prog_name is a string to indicate the program whose MAIN is being called; when *rout_name* is specified, it is the owner of the being called routine;
rout_name is the being called routine;
arg_val are the argument passed to the being called routine, if there are any.

Examples:

```
CALLS ('pippo') -- Execution jumps to the BEGIN of the MAIN
      -- part of pippo program
CALLS ('pippo', 'f1') -- Execution jumps to the BEGIN of f1
      -- routine, owned by pippo program
CALLS ('pippo', 'f1')(1,5.2) -- Execution jumps to the BEGIN
      -- of f1 routine, owned by pippo
      -- program, and there are two
      -- routine parameters.
```

See also:

[par. 7.4 Passing Arguments on page 142](#).

10.8 CANCEL Statement

The CANCEL statement has different forms for canceling motion, SEMAPHOREs, or the system ALARM state. The CANCEL motion statements cancel the current motion or all motions for a specific arm, list of arms, or all arms. Another option is to cancel the current path segment or all path segments for a specific arm, list of arms, or all arms.

Syntax:

```
CANCEL || ALL | CURRENT|| <SEGMENT> <FOR || ARM[n] <,
ARM[n]>... | ALL ||>
CANCEL ALARM
```

`CANCEL semaphore_var`

Comments:

Canceling a motion causes the arm to decelerate smoothly until the motion ceases. A canceled motion cannot be resumed.

`CANCEL CURRENT` cancels only the current motion. If there is a pending motion when the statement is issued, it is executed immediately. If the current motion is a path motion, all path segments are canceled. `CANCEL ALL` cancels both the current motion and any pending motions.

`CANCEL CURRENT SEGMENT` cancels only the current path segment. If additional nodes remain in the path when the `CANCEL CURRENT SEGMENT` statement is executed, they are processed immediately. If there are no remaining nodes in the path or the `CANCEL ALL SEGMENT` statement was used, pending motions (if any) are executed immediately. `CANCEL ALL SEGMENT` is equivalent to canceling the current motion using `CANCEL CURRENT`.

The programmer can specify an arm, a list of arms, or all arms in a `CANCEL` motion statement. If nothing is specified, motion for the default arm is canceled.

`CANCEL ALARM` clears the alarm state of the controller. It has the same effect as the SHIFT SCRN key on the keyboard. This statement will not execute properly if the controller is in a fatal state.

`CANCEL semaphore_var` clears all unused signals. The signal counter is set to zero. This statement should be executed at the beginning of the program to clear out any outstanding signals from a previous execution.

If programs are currently waiting on a SEMAPHORE, the `CANCEL semaphore_var` statement will result in an error.

The `CANCEL` statement is permitted as a condition handler action. Refer to the [Condition Handlers](#) chapter for further information.

Examples:

```
CANCEL ALL
CANCEL ALL SEGMENT
CANCEL ALL FOR ARM[1], ARM[2]
CANCEL ALL SEGMENT FOR ARM[1]
CANCEL ALL FOR ALL
```

```
CANCEL CURRENT
CANCEL CURRENT SEGMENT
CANCEL CURRENT FOR ARM[3]
CANCEL CURRENT SEGMENT FOR ARM[3], ARM[2]
CANCEL CURRENT FOR ALL
```

`CANCEL ALARM`

`CANCEL resource`

See also:

- [LOCK Statement](#)
- [HOLD Statement](#)
- [SIGNAL Statement](#)
- [WAIT Statement](#)
- [Motion Control Chapter](#)

10.9 CLOSE FILE Statement

The CLOSE FILE statement closes a LUN, ending the connection between the program and the device.

Syntax:

```
CLOSE FILE || lun_var | ALL ||
```

Comments:

lun_var can be any user-defined INTEGER variable that represents an open LUN. A LUN can be closed only by the program that opened it. All LUNs used by a program are closed automatically when program execution is completed or the program is deactivated.

Using the ALL option closes all LUNs that were opened by that program. LUNs opened by other programs are not affected.

Any buffered data is written to the device before the CLOSE FILE statement is executed.

Examples:

```
CLOSE FILE file_lun
```

```
CLOSE FILE crt1_lun
```

```
CLOSE FILE ALL lun_var
```

See also:

[OPEN FILE Statement](#)

[Communication Chapter](#)

10.10 CLOSE HAND Statement

The CLOSE HAND statement closes a hand (too).

Syntax:

```
CLOSE HAND number <FOR || ARM[n] <, ARM[n]>... | ALL ||>
```

Comments:

number indicates the number of the hand to be closed. Two hands are available per arm.

The effect of the close operation depends on the type of hand being operated (refer to [Motion Control](#) chapter).

The optional FOR ARM clause can be used to indicate a particular arm, list of arms, or all arms. If not specified, the default arm is used.

Examples:

```
CLOSE HAND 1
```

```
CLOSE HAND 2 FOR ARM[2]
```

See also:

[OPEN FILE Statement](#)

[RELAX HAND Statement](#)

[Motion Control Chapter](#)

10.11 CONDITION Statement

The CONDITION statement defines a condition handler in the executable section of a program.

Syntax:

```

CONDITION      [int_expr]      <FOR      ARM [n] >      <NODISABLE>
<ATTACH><SCAN (number) >:
  WHEN cond_expr DO
    action_list
  <WHEN cond_expr DO
    action_list>...
ENDCONDITION

```

Comments:

The programmer identifies each condition handler by *int_expr*, an INTEGER expression that can range from 1 to 255.

Conditions to be monitored are specified in the condition expression, *cond_expr*, as explained in the [Condition Handlers](#) chapter.

The *action_list* specifies the actions to be taken when the condition handler is triggered (condition expression becomes TRUE), as explained in the [Condition Handlers](#) chapter.

Condition handlers must be enabled to begin monitoring.

The optional FOR ARM clause can be used to designate a particular arm for the condition handler. Globally enabled motion events will apply to any moving arm while local events will apply to the arm of the MOVE statement they are associated with (via the WITH option). If the FOR ARM clause is not included, the arm specified by the PROG_ARM attribute on the PROGRAM statement is used. If PROG_ARM is not specified, the value of the predefined variable \$DFT_ARM is used. The arm is used for arm related conditions and actions.

The optional NODISABLE clause indicates the condition handler will not automatically be disabled when the condition handler is triggered. The NODISABLE clause is not allowed on condition handlers that contain state conditions.

The optional ATTACH clause causes the condition handler to be attached immediately after it is defined. If not specified, the condition handler can be attached elsewhere in the program using the ATTACH statement.

When a condition handler is attached, only code belonging to the same program containing the ATTACH can enable, disable, purge, detach, or redefine the condition handler. The DETACH statement is used to release exclusive control of a condition handler.

The optional SCAN clause can be used to indicate a larger monitoring period, respect to the scan rate defined in the system (\$TUNE[1]), for conditions that are states.

Examples:

```

CONDITION [14] NODISABLE ATTACH:
  WHEN PAUSE DO
    $DOUT[21] := OFF
ENDCONDITION

CONDITION [23] FOR ARM [2] :
  WHEN DISTANCE 60 BEFORE END DO -- applies to arm 2
    LOCK -- applies to arm 2
  WHEN $ARM_DATA[4].PROG_SPD_OVR > 50 DO
    LOCK ARM [4]

```

```

ENDCONDITION
-- This CONDITION is scanned every $TUNE[1]*2 milliseconds
CONDITION[55] SCAN(2):
WHEN $DOUT[20]=TRUE DO
    PAUSE
ENDCONDITION

```

See also:

[ENABLE CONDITION Statement](#)
[DISABLE CONDITION Statement](#)
[PURGE CONDITION Statement](#)
[ATTACH Statement](#)
[DETACH Statement](#)
[Condition Handlers Chapter](#)

10.12 CONST Statement

The CONST statement marks the beginning of a constant declaration section in a program or routine.

Syntax:

```

CONST
    name = || literal | predef_const_id ||
<name = || literal | predef_const_id ||>...

```

Comments:

A constant declaration establishes a constant identifier with a name and an unchanging value.

name can be any user-defined identifier.

The data type of the constant is determined by its value, as follows:

- INTEGER (whole number)
- REAL (decimal point or scientific notation)
- BOOLEAN (TRUE, FALSE, ON, or OFF)
- STRING (enclosed in single quotes)

The value can be specified as a literal or predefined constant.

The translator checks to make sure the values are legal. Expressions are not allowed.

The constant declaration section is located between the PROGRAM or ROUTINE statement and the corresponding BEGIN statement.

Examples:

```

PROGRAM example
CONST
    temp = 179.04      -- REAL
    time = 300         -- INTEGER
    flag = ON          -- BOOLEAN
    movement = LINEAR -- INTEGER
    part_mask = 0xF3   -- INTEGER
BEGIN
    .
    .
    .

```

```
END example
```

See also:

[TYPE Statement](#)

[VAR Statement](#)

[Data Representation Chapter](#)

10.13 CYCLE Statement

The CYCLE statement allows the programmer to create a continuous cycle.

Syntax:

```
CYCLE <frequency>
```

Comments:

<frequency> is an optional integer parameter to specify every how many milliseconds the program cycle is run. This is to avoid the program to take a too large amount of the CPU time.

Examples of using <frequency> optional parameter: CYCLE 10 or CYCLE myVar or CYCLE 1+2+3 .

Only one CYCLE statement is allowed in the program and it must be in the main program.

The CYCLE statement is not allowed in routines.

The CYCLE statement is not allowed if the CYCLE option is used on the BEGIN statement.

The difference between the CYCLE statement and the CYCLE option is that using the CYCLE statement permits some initialization code that is not executed each cycle.

When the program END statement is encountered, execution continues back at the CYCLE statement. This continues until the program is deactivated or an EXIT CYCLE statement is executed. When an EXIT CYCLE statement is executed, the current cycle is terminated and execution immediately continues back at the CYCLE statement.

Examples:

```
PROGRAM example1
  -- declaration section
BEGIN
  -- initial executable
  -- section
CYCLE 100
  -- cycled executable
  -- section, run every 100 ms
END example1
```

```
PROGRAM example2
  -- declaration section
BEGIN CYCLE
  -- executable section
END example2
```

See also:

[BEGIN Statement](#)

[EXIT CYCLE Statement](#)

10.14 DEACTIVATE Statement

The DEACTIVATE statement deactivates programs that are in the running, ready, paused, or paused-ready states.

Syntax:

```
DEACTIVATE < | | prog_name <,prog_name>... | ALL | |>
```

Comments:

If *prog_name* or a list of names is specified, those programs are deactivated. If no *prog_name* is specified, the program issuing the statement is deactivated. If ALL is specified, all executing programs are deactivated.

When a program is deactivated, the following occurs for that program:

- current and pending motions are canceled
- condition handlers are purged
- the program is removed from any lists (semaphores)
- reads, pulses, and delays are canceled
- current and pending system calls are aborted
- opened files are closed
- attached resources are detached
- locked arms are unlocked but the motion still needs to be resumed

Deactivated programs remain loaded, but do not continue to cycle and cannot be resumed. They can be reactivated with the ACTIVATE statement.

The DEACTIVATE statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
DEACTIVATE
```

```
DEACTIVATE weld_main, weld_util
```

```
DEACTIVATE ALL
```

See also:

[ACTIVATE Statement](#)

10.15 DECODE Statement

The DECODE statement converts a string into individual values.

Syntax:

```
DECODE (string_expr, var_id <, var_id>...)
```

Comments:

The STRING value represented by *string_expr* is converted into individual values that are assigned to the corresponding *var_id*.

The data type of the value is determined by the data type of each *var_id*. The STRING value must be able to convert to this data type or a trappable error will result.

Valid data types are:

BOOLEAN	INTEGER
JOINTPOS	POSITION
REAL	STRING
VECTOR	XTNDPOS

var_id can also be a predefined variable reference as long as the predefined variable does not have value limits. Refer to [Predefined Variables List](#) chapter to determine if a predefined variable has limits.

Optional [Format Specifiers](#) can be used with *var_id* as they are used in a [READ Statement](#).

Examples:

```
PROGRAM sample
VAR
  s, str : STRING[10]
  i, x, y, z : INTEGER
BEGIN
  READ(str) -- assume 4 6 8 is entered
  DECODE(str, x, y, z) -- x = 4, y = 6, z = 8
  DECODE('1234abcd', i, s) -- generates i = 1234, s = abcd
  DECODE('-1234abcd', i) -- generates i = -1234
  DECODE(+, i) -- trappable error
  READ(str) -- assume 101214 is entered
  DECODE(str, x::3, y::1, z::2) -- x = 101, y = 2, z = 14
END sample
```

See also:

[ENCODE Statement](#)
[READ Statement](#)
[Communication Chapter](#)

10.16 DELAY Statement

The DELAY statement causes execution of the program issuing it to be suspended for a specified period of time.

Syntax:

```
DELAY int_expr
```

Comments:

int_expr indicates the time, in milliseconds, to delay.

The following events continue even while a program is delayed:

- current and pending motions
- condition handler scanning
- current output pulses
- current and pending system calls

DELAY is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

Examples:

```
MOVE TO pickup
DELAY 200
CLOSE HAND 1
```

See also:

[WAIT FOR Statement](#)

10.17 DETACH Statement

The DETACH statement releases an attached resource. The statement can be applied to arms, I/O devices, condition handlers, or timers.

Syntax:

```
DETACH || ARM <ALL> | ARM[n] <, ARM[n]>... ||  
DETACH device_str <, device_str>...  
DETACH CONDITION || ALL | [number] <, CONDITION [number]> ...  
||  
DETACH $TIMER [n] <, $TIMER [n]>...
```

Comments:

When an arm is detached, other programs will be permitted to cause motion on the arm.

When the ARM ALL option is used, all arms currently attached to the program executing the DETACH statement are detached.

It is only a warning if a program attempts to detach an arm that is currently not attached to any program. An error occurs if a program attempts to detach an arm that is currently attached by another program.

The *device_str* can be any I/O device name including the following predefined window and communication devices:

Communication Devices:	Window Devices:
COM1:	CRT:
COM2:	CRT1:
COM3:	CRT2:
COM4:	CRT3:
COMP:	TP:
DGM0:	TP0:
DGM1:	TP1:
DGM2:	TP2:
DGM3:	TP3:
DGM4:	

When an I/O device is detached, other programs can open a LUN on that device and if it is a window device, the window related built-ins can be applied to that device.

An error occurs if a program attempts to detach a device that is currently attached by another program.

When a condition handler is detached, other programs can enable, disable, purge, or redefine the condition handler.

When a timer is detached, other programs will be permitted read and write access to that timer.

The DETACH statement used to detach a condition handler or timer must be contained in the same program owning the corresponding ATTACH statement. An error occurs if the program attempts to execute a DETACH statement contained in a different program. It is also an error if a program attempts to detach a condition handler that does not exist.

When a program terminates or is deactivated, all attached resources are automatically detached.

The DETACH statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
DETACH COM1: -- detach a communication device
DETACH CRT2: -- detach a window device
```

```
DETACH ARM -- detach the default arm
DETACH ARM[1], ARM[3]
```

```
DETACH CONDITION[6] -- must be in same program owning the
ATTACH
DETACH CONDITION ALL
```

```
DETACH $TIMER[1] -- must be in same program owning the ATTACH
DETACH $TIMER[8], $TIMER[9]
```

See also:

[ATTACH Statement](#)

[PROGRAM Statement \(DETACH Attribute\)](#)

[PROGRAM Statement \(ATTACH Attribute\)](#)

10.18 DISABLE CONDITION Statement

The DISABLE CONDITION statement disables a globally enabled condition handler.

Syntax:

```
DISABLE CONDITION || ALL | [int_exp] <, CONDITION
[int_exp]>... ||
```

Comments:

int_exp is the number of the condition handler to be disabled.

Disabled condition handlers can be re-enabled.

Condition handlers are disabled automatically when the condition expression is triggered unless the NODISABLE clause is included in the definition.

If the ALL option is used, all of the program's globally enabled condition handlers are disabled.

If a condition handler is also currently enabled as part of a WITH clause, it will be disabled when the MOVE finishes, not when the DISABLE CONDITION statement is executed.

It is an error if the program attempts to disable a condition handler that is currently attached in another program.

The DISABLE CONDITION statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
DISABLE CONDITION[2]
```

```
DISABLE CONDITION[error_chk], CONDITION[signal_chk]
```

```
DISABLE CONDITION ALL
```

See also:

[CONDITION Statement](#)
[ENABLE CONDITION Statement](#)
[MOVE Statement](#)
[PURGE CONDITION Statement](#)
[Condition Handlers Chapter](#)

10.19 DISABLE INTERRUPT Statement

DISABLE INTERRUPT Statement disables any possible incoming interrupt of the current thread of execution.

Syntax:

```
DISABLE INTERRUPT
    <statements>
ENABLE INTERRUPT
```

Comments:

statements included between the DISABLE INTERRUPT and the ENABLE INTERRUPT are prevented from interruption by Interrupt Service Routines occurring in the same program. Those routines will be interpreted after the ENABLE INTERRUPT statement is executed.

Only the following subset of PDL2 statements is allowed between the DISABLE INTERRUPT and the ENABLE INTERRUPT instructions: ATTACH and DETACH of arms, timers, conditions; OPEN, CLOSE and RELAX HAND; RETURN; ENABLE, DISABLE, PURGE CONDITION; SIGNAL EVENT, SEGMENT, semaphore; CANCEL ALARM, motion, semaphore; RESUME; LOCK and UNLOCK; HOLD; assignement, increment and decrement of variables.

This statement should only be used when really necessary so to avoid the nesting of thread levels in the program call chain. It is important to specify a minimum amount of statements for not compromising a correct program interpretation.

```
PROGRAM example
ROUTINE user_isr
BEGIN
    -- Interrupt Service routine statements
    <statements....>
    DISABLE INTERRUPT
        ENABLE CONDITION[13]
        $BIT[100] := ON
        RETURN
    ENABLE INTERRUPT
END user_isr
BEGIN -- main
    CONDITION[13] :
        WHEN $DIN[18] DO
            user_isr
    ENDCONDITION
    ENABLE CONDITION[13]
    -- Main statements
    <statements..>
END example
```

See also:

[Condition Handlers chapter.](#)

10.20 ENABLE CONDITION Statement

The ENABLE CONDITION statement globally enables a condition handler.

Syntax:

```
ENABLE CONDITION [int_expr] <, CONDITION [int_expr]>...
```

Comments:

int_expr is the number of the condition handler to be enabled.

A condition expression is monitored only when the condition handler is enabled.

If the specified condition handler is also currently enabled as part of a WITH clause, it will remain enabled when the motion finishes, instead of being disabled.

It is an error if the program attempts to enable a condition handler that is currently attached in another program.

The ENABLE CONDITION statement is permitted as a condition handler action. Refer to [Condition Handlers chapter](#) for further information.

Examples:

```
ENABLE CONDITION [3]
```

See also:

[CONDITION Statement](#)
[DISABLE CONDITION Statement](#)
[MOVE Statement](#)
[PURGE CONDITION Statement](#)
[Condition Handlers Chapter](#)

10.21 ENCODE Statement

The ENCODE statement converts individual values into a STRING.

Syntax:

```
ENCODE (string_id, expr <, expr>...)
```

Comments:

The expressions represented by *expr* are converted into a string value and assigned to *string_id*.

Valid data types for the expressions are:

BOOLEAN	INTEGER
JOINTPOS	POSITION
REAL	STRING
VECTOR	XTNDPOS

Optional [Format Specifiers](#) can be used with *expr* as they are used in a [WRITE Statement](#). Note that, if *expr* is not left justified, the first character in *string_id* is a blank one.

The maximum length of the STRING is determined by the declared length of *string_id*. If the new STRING is longer than the declared length, then the new STRING is truncated to fit into *string_id*.

Examples:

```

PROGRAM sample
VAR
    str : STRING[20]
    x,y : INTEGER
    z : REAL
BEGIN
    x := 23
    y := 1234567
    z := 32.86
    ENCODE (str, x, y, z)
    WRITE (str) -- outputs '23' '1234567' '32.860'
    ENCODE (str, x::4, y::8, z::4::1)
    WRITE (str) -- outputs '23' '123456732.9'
END sample

```

See also:

[DECODE Statement](#)
[WRITE Statement](#)
[Communication Chapter](#)

10.22 END Statement

The END statement marks the end of a program or routine.

Syntax:

```
END name
```

Comments:

name is the user-defined identifier used to name the program or routine.

If the END is in a procedure routine, it returns program control to the calling program or routine.

It is an error if the END of a function routine is executed. A function routine must execute a RETURN statement to return program control to the calling program or routine.

If the CYCLE option was used with the BEGIN statement or the CYCLE statement is used in the program, the END statement transfers control to the CYCLE and the statements between CYCLE and END execute again. This keeps a program running until it is deactivated.

Examples:

```

PROGRAM example
    -- declaration section
BEGIN
    -- executable section
END example

```

```

ROUTINE r1
    -- local decalration section
BEGIN
    --routine executable section
END r1

```

See also:

[BEGIN Statement](#)
[CYCLE Statement](#)
[EXIT CYCLE Statement](#)
[RETURN Statement](#)

10.23 EXIT CYCLE Statement

The EXIT CYCLE statement causes program execution to skip the remainder of the current cycle and immediately begin the next cycle.

Syntax:

```
EXIT CYCLE <|| prog_name <,prog_name>... | ALL ||>
```

Comments:

If *prog_name* or a list of names is specified, those programs exit their current cycles. If no name is included, the program issuing the statement exits its current cycle. If ALL is specified, all executing programs exit their current cycles.

Exiting a cycle will cancel all current and pending motion, all asynchronous statements such as DELAY, PULSE, SYS_CALL, etc., and most of program stack system variables (apart from \$CYCLE, \$PROG_CONDS, \$PROG_NAME, \$PROG_CNFG) are reset.

Exiting a cycle will NOT close files, detach resources, reset \$HDIN, disable or purge condition handlers, or unlock arms. Therefore, the CYCLE statement should be placed after these types of statements in order to prevent duplication on successive cycles.

If the program has not executed a CYCLE statement before the EXIT CYCLE statement is executed, and the CYCLE option is not on the BEGIN, an error occurs.

An exited cycle cannot be resumed.

The EXIT CYCLE statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
EXIT CYCLE
```

```
EXIT CYCLE weld_prog, weld_util
```

```
EXIT CYCLE ALL
```

See also:

[BEGIN Statement](#)
[CYCLE Statement](#)

10.24 FOR Statement

The FOR statement executes a sequence of statements a known number of times.

Syntax:

```
FOR intvar := startval || TO | DOWNTO || endval <STEP
  step_val> DO
    <statement>...
  ENDFOR
```

Comments:

intvar is initially assigned the value of *startval*. It is then incremented or decremented by a stepping value each time through the loop until *endval* is either reached or exceeded. The loop consists of executable statements between the FOR and ENDFOR.

startval, *endval*, and *step_val* can be any INTEGER expression. *intvar* must be an INTEGER variable.

If TO is used *intvar* is incremented. *startval* must be less than or equal to *endval* or the loop will be skipped.

If DOWNTO is used *intvar* is decremented. *startval* must be greater than or equal to *endval* or the loop will be skipped.

If the STEP option is specified, the stepping value is indicated by *step_val*. The stepping value used should be a positive INTEGER value to ensure that the loop is interpreted correctly. If no STEP value is specified, a value of one is used.

When the stepping value is one, the statements are executed the absolute value of (*endval* - *startval* + 1) times. If *startval* equals *endval*, the loop is executed one time.

A GOTO statement should not be used to jump into, or out of, a FOR loop. If the ENDFOR is reached without having executed the corresponding FOR statement, an error will occur. If a GOTO is used to jump out of a FOR loop after the FOR statement has been executed, information pertaining to the FOR statement execution will be left on the stack.

Examples:

```
FOR i := 21 TO signal_total DO
    $DOUT[i] := OFF
ENDFOR
```

See also:

[WHILE Statement](#)
[REPEAT Statement](#)

10.25 GOTO Statement

The GOTO statement unconditionally transfers program control to the place in the program specified by a statement label.

Syntax:

```
GOTO statement_label
```

Comments:

statement_label is a label identifier, at the left margin, followed by two consecutive colons (::).

Executable statements may follow on the same line as, or on any line after, the statement label.

The label must be within the same ROUTINE or PROGRAM body as the GOTO statement or a translator error will occur.

A GOTO statement should not be used to jump into, or out of, a FOR loop. If the ENDFOR is reached without having executed the corresponding FOR statement, an error will occur. If a GOTO is used to jump out of a FOR loop after the FOR statement has been executed, information pertaining to the FOR statement execution will be left on the stack.

Examples:

```
PROGRAM example
BEGIN
    .
    .
    .
    IF (error) THEN
        GOTO err_prt
    ENDIF
    .
    .
```

```
err_prt:: WRITE(This is where the GOTO transfers to. )
END example
```

See also:

[FOR Statement](#)
[IF Statement](#)
[REPEAT Statement](#)
[SELECT Statement](#)
[WHILE Statement](#)

10.26 HOLD Statement

The HOLD statement places all running holdable programs in a ready state and causes motion to decelerate to a stop.

Syntax:

```
HOLD
```

Comments:

The HOLD statement works exactly like the HOLD button on the TP and control panel.

A HOLD causes the arm to decelerate smoothly until the motion ceases. The predefined variable \$HLD_DEC_PER indicates the rate of deceleration.

START button must be pressed to place holdable programs back into a running state.

The HOLD statement can be used in both holdable and non-holdable programs.

HOLD is an asynchronous statement which means it continues while the program is paused and condition handlers continue to be scanned. However, if a condition handler triggers while a program is held, the actions are not performed until the program is unheld.

The HOLD statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
HOLD
```

See also:

[CANCEL Statement](#)
[LOCK Statement](#)
[PAUSE Statement](#)

10.27 IF Statement

The IF statement is used to choose between two possible courses of action, based on the result of a Boolean expression.

Syntax:

```
IF bool_expr THEN
  <statement>...
<ELSE
  <statement>... >
ENDIF
```

Comments:

bool_expr is any expression that yields a Boolean result.

If *bool_expr* is TRUE, the statement(s) following the IF clause are executed. If *bool_expr* is FALSE, program control is transferred to the first statement after the ENDIF.

If the ELSE clause is specified and *bool_expr* is FALSE, the statement(s) between the ELSE and ENDIF are executed.

Examples:

```
IF (file_error) THEN
    WRITE ('***ERROR*** data file not found.')
ELSE
    WRITE ('Loading data. . .')
ENDIF
```

See also:

[SELECT Statement](#)

10.28 IMPORT Statement

The IMPORT statement imports, from an external program, any identifiers which have been declared with the GLOBAL attribute (see [par. 3.2.4.2 GLOBAL attribute and IMPORT statement on page 68](#)); this statement allows the user to import GLOBAL identifiers without having to explicitly declare each of them.

Syntax:

```
IMPORT prog_name
```



NOTE THAT:

the IMPORT clause must be after the PROGRAM clause and before any declaration clause.

In general the order is:

- PROGRAM
- IMPORT
- TYPE
- VAR
- ROUTINES
- code.

Comments:

Importing from a program, causes the current program to be able to make use of any GLOBAL identifiers (types, variables and routines) belonging to another program.

prog_name is a STRING representing the name of the owning program (program which owns the being imported GLOBAL variables and/or routines). It can also contain a relative or absolute path.



Local variables of a global routine cannot be declared with the GLOBAL attribute.

Examples:

```
IMPORT 'tv_move'
IMPORT 'sys\util\tv_move'
```

See also:

- [par. 3.2.4 Shared types, variables and routines on page 67](#)

10.29 LOCK Statement

The LOCK statement suspends motion for a specific arm, list of arms, or all arms.

Syntax:

```
LOCK || <ARM[n] <,ARM[n]>...> | ALL ||
```

Comments:

Locking an arm causes the arm to decelerate smoothly until the motion ceases. The predefined variable \$HLD_DEC_PER indicates the rate of deceleration. LOCK prevents pending motions or new motions from starting on the locked arm.

The programmer can specify an arm, a list of arms, or all arms to be locked. If nothing is specified, the default arm is locked.

To unlock an arm, an UNLOCK statement must be issued from the same program that issued the LOCK. After an UNLOCK statement has been issued, motion can be resumed by issuing a RESUME statement.

Motion can be canceled using CANCEL motion statement while the arm is locked.

The LOCK statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
LOCK
```

```
LOCK ALL
```

```
LOCK ARM[2], ARM[5]
```

```
CONDITION[3] :
```

```
  WHEN $DIN[3]+ DO
    LOCK ARM[3]
  ENDCONDITION
```

See also:

[UNLOCK Statement](#)
[RESUME Statement](#)
[CANCEL Statement](#)

10.30 MOVE Statement

The MOVE statement controls arm motion. Different clauses and options allow for many different kinds of motion, as described in [Chap.4. - Motion Control](#).

Syntax:

```
MOVE    <ARM[n]>    <trajectory>    dest_clause    <opt_clauses>
      <sync_clause>
```

Comments:

The *dest_clause* specifies the kind of move and its destination. It can be any of the following:

```
TO || destination | joint_list || <VIA_clause>
NEAR destination BY distance
AWAY distance
RELATIVE vector IN frame
```

```

ABOUT vector BY distance IN frame
BY relative_joint_list
FOR distance TO destination
    
```

destination in the above clauses can be a POSITION, JOINTPOS, XTNDPOS, or node expression. If it is a node, the standard node fields of that node are used in the motion.

joint_list is a list of real expressions, with each item corresponding to the joint angle of the arm being moved *frame* in the above clauses must be one of the predefined constants BASE, TOOL, or UFRAME

The optional arm clause (*ARM[n]*) designates the arm to be moved. The designated arm is used for the entire MOVE statement. If the ARM clause is not included, the default arm is moved.

The optional *trajectory* clause designates a trajectory for the move. It can be any of the following predefined constants:

```

JOINT
LINEAR
CIRCULAR
    
```

If the *trajectory* clause is not included, the value of \$MOVE_TYPE is used.

opt_clauses provide more detailed instructions for the motion. Optional clauses include the following:

```

ADVANCE
TIL cond_expr <, TIL cond_expr>...
WITH designations <, designations>...
    
```

If both the TIL and WITH clauses are specified, all TIL clauses must be specified before the WITH clauses.

The *sync_clause* allows two arms to be moved simultaneously using SYNCMOVE. The arms start and stop together. The optional WITH and TIL clauses can be included as part of a SYNCMOVE clause.

The *TIL clause* can only be specified on the MOVE side.

MOVEFLY can be used in place of the reserved word MOVE to specify continuous motion. To execute the FLY, the ADVANCE clause must also be included in the MOVEFLY statement. If a *sync_clause* is specified, it must be SYNCMOVEFLY, and the ADVANCE clause must be part of the MOVEFLY.

If a MOVE statement needs more than a single line, commas must be used to end a line after the *dest_clause* or after each optional clause. The reserved word ENDMOVE must then be used to indicate the end of the statement. Refer to the examples below.

MOVE is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

Examples:

```

MOVE NEAR pick_up BY 200.0
MOVE TO pick_up
MOVE AWAY 400.0
MOVE RELATIVE VEC(100, 0, 100) IN TOOL
MOVE ABOUT VEC(0, 100, 0) BY 90 IN BASE
MOVE BY {alpha, beta, gamma, delta, omega}
    
```

```

MOVE FOR dist TO destination

MOVE ARM[1] TO destination
MOVE JOINT TO pick_up
MOVE CIRCULAR TO destination VIA arc

MOVE ARM[1] TO part SYNCMOVE ARM[2] TO front
MOVEFLY TO middle ADVANCE
MOVE TO end_part

MOVE NEAR slot BY 250 ADVANCE
OPEN HAND 1
MOVE TO slot

MOVE TO flange WITH CONDITION[2], CONDITION[3],
      WITH $PROG_SPD_OVR = 50, CONDITION[4],
ENDMOVE
MOVE TO pickup
TIL $DIN[part_sensor]+,
      WITH CONDITION[1],
      WITH $PROG_SPD_OVR = 50,
ENDMOVE

MOVE ARM[1] TO slot WITH $PROG_SPD_OVR = 100,
SYNCMOVE ARM[2] TO front WITH $LIN_SPD = 200,
ENDMOVE

MOVE TO slot TIL $DIN[1]+

MOVEFLY ARM[1] TO slot ADVANCE SYNCMOVEFLY ARM[2] TO front

MOVE ARM[1] TO part,
      TIL $DIN[1]+,
      WITH CONDITION[1], -- applies to arm 1
      SYNCMOVE ARM[2] TO front,
      WITH CONDITION[2], -- applies to arm 2
ENDMOVE

```

See also:

[MOVE ALONG Statement](#)
[Chap.4. - Motion Control](#)

10.31 MOVE ALONG Statement

The MOVE ALONG statement specifies arm movement along the nodes defined for a PATH variable.

Syntax:

MOVE <ARM[n]> ALONG *path_var*<.NODE [node_range]> <opt_clauses>

Comments:

The optional arm clause (ARM[n]) designates the arm to be moved. The designated arm is used for the entire MOVE ALONG statement. If the ARM clause

is not included, the default arm is moved.

The arm applies to the main destination field (\$MAIN_POS, \$MAIN_JNT, \$MAIN_XTND) only. If the main destination is a JOINTPOS (\$MAIN_JNT) or an XTNDPOS (\$MAIN_XTND), the arm number must match the one used in the NODEDEF definition.

If .NODE[node_range] is not present, motion proceeds to the first node of *path_var*, then to each successive node until the end of *path_var* is reached.

If .NODE[node_range] is present, the arm can be moved along a range of nodes specified within the brackets. The range can be in the following forms:

- [n..m] Motion proceeds to node n of *path_var*, then to each successive node until node m of *path_var* is reached. Backwards motion is allowed by specifying node n greater than node m.
- [n..] Motion proceeds to node n of *path_var*, then to each successive node until the end of *path_var* is reached.

Opt_clauses provide more detailed instructions for the motion. Optional clauses include the following:

ADVANCE

WITH designations <, designations>...

MOVEFLY can be used in place of the reserved word MOVE to specify continuous motion. To execute the fly, the ADVANCE clause must also be included with the MOVEFLY statement. Specifying MOVEFLY applies to the end of the path, not for each node. In addition, \$TERM_TYPE applies to the end of the path and not for each node.

If a MOVE ALONG statement needs more than a single line, commas must be used to end a line after the *path_var* or after each optional clause. The reserved word ENDMOVE must then be used to indicate the end of the statement. Refer to the examples below.

MOVE ALONG is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

Examples:

```
MOVE ALONG pth
MOVE ALONG pth.NODE [1..5]
MOVE ARM [2] ALONG pth
```

```
MOVE ALONG pth,
    WITH $PROG_SPD_OVR = 50,
ENDMOVE
```

See Also:

[MOVE Statement](#)
[Chap.4. - Motion Control](#)

10.32 OPEN FILE Statement

The OPEN FILE statement opens a LUN on the specified device. This establishes a

connection between the program and the device through which I/O operations can be performed.

A user-defined integer variable is used by PDL2 to represent the LUN. That variable can be used in subsequent READ and WRITE statements and in built-in routines requiring LUN parameters to indicate the device on which an I/O operation is to be performed.

A LUN remains opened until it is closed with a CLOSE FILE statement, program execution is completed, or the program is deactivated.

The syntax of the OPEN FILE statement is as follows:

```
OPEN FILE lun_var (device_str, access_str) <with_clause>
```

The *lun_var* can be any user-defined integer variable.

The *device_str* can be any string expression representing an I/O device (a window, communication, or file device). File devices also include a file name and file extension. The default file device is 'UD:'.

The *access_str* can be any string expression representing the access with which the device is to be opened. The following types of access are allowed:

R	-- read only
W	-- write only
RW	-- read and write
WA	-- write append
RWA	-- read and write append

If a file is opened with write access (W or RW), other programs will be denied access to that file. If a file is opened with read only access (R), other programs will also be allowed to open that file with read only access. Write operations will be denied. If a file already exists on the device and an OPEN FILE on that file is done, the same file is opened and its contents can be added to if 'RWA' was specified. If opened with the 'RW' attributes, the file must be present on the device upon which the file is being opened.

Examples of the OPEN FILE statement follow:

```
OPEN FILE crt1_lun ('CRT1:', 'RW') -- opens window CRT1:, read and write
OPEN FILE file_lun ('stats.dat', 'R') --opens stats.dat,read only
OPEN FILE comm_lun ('COM1:', 'R') --opens port COM1:, read only
OPEN FILE win_lun ('abcd:', 'RW') --opens user-defined window ABCD:
```

See also:

- [DV_CNTRL Built-In Procedure](#)
- [CLOSE FILE Statement](#)
- [READ Statement](#)
- [WRITE Statement](#)
- [Chap.9. - Communication](#)

10.32.1 WITH Clause

The optional WITH clause can designate temporary values for predefined variables related to LUNs. The WITH clause affects only the LUN currently being opened.

The syntax of the WITH clause is as follows:

```
WITH predef_lun_var = value <, predef_lun_var = value>...
```

The following predefined variables can be used in a WITH clause:

\$FL_ADLMT	\$FL_NUM_CHARS
\$FL_BINARY	\$FL_PASSALL
\$FL_DLMT	\$FL_RANDOM
\$FL_ECHO	\$FL_RDFLUSH
\$FL_SWAP	

For example:

```
OPEN FILE file_lun (stats.log, R) WITH $FL_BINARY = TRUE
```

If a statement needs more than a single line, a comma is used to end the OPEN FILE line. Each new line begins with the reserved word WITH and ends with a comma. The reserved word ENDOPEN must be used to indicate the end of the OPEN FILE statement if it spans more than one line.

For example:

```
OPEN FILE file_lun ('stats.log', 'R'),
    WITH $FL_SWAP = TRUE, $FL_BINARY = TRUE,
ENDOPEN
```

```
OPEN FILE comm_lun ('COM1:', 'RW') WITH $FL_SWAP = TRUE,
    WITH $FL_BINARY = TRUE,
    -- Delimiters of ctrlc, down, up, enter, prev keys
    WITH $FL_ADLMT = '\010\013\011\027\003',
ENDOPEN
```

Examples:

```
OPEN FILE crt1_lun ('CRT1:', 'RW')
OPEN FILE com_port ('COM1:', 'RW')
```

```
OPEN FILE file_lun (stats.log, 'R'),
    WITH $FL_SWAP = TRUE,
    WITH $FL_BINARY = TRUE,
ENDOPEN
```

10.33 OPEN HAND Statement

The OPEN HAND statement opens a hand (tool).

Syntax:

```
OPEN HAND number <FOR || ARM[n] <,ARM[n]>... | ALL || >
```

Comments:

number indicates the number of the hand to be opened. Two hands are available per arm.

The effect of the open operation depends on the type of hand being operated (refer to [Chap.4. - Motion Control](#)) configured using the HAND configuration tool delivered with the system software.

The effect of the open operation depends on the type of hand being operated (refer to [Chap.4. - Motion Control](#)).

The optional FOR ARM clause can be used to indicate a particular arm, a list of

arms, or all arms. If not specified, the default arm is used.

Examples:

```
OPEN HAND 1
```

```
OPEN HAND 2 FOR ARM[2]
```

See also:

[CLOSE HAND Statement](#)

[RELAX HAND Statement](#)

[Chap.4. - Motion Control](#)

10.34 PAUSE Statement

The PAUSE statement causes a program to be paused until an UNPAUSE operation is executed for that program.

Syntax:

```
PAUSE < || prog_name <, prog_name>... | ALL || >
```

Comments:

If *prog_name* or a list of names is specified, those programs are paused. If no name is included, the program issuing the statement is paused. If ALL is specified, all running programs are paused.

Pausing a program in the running state places that program in a paused state. Pausing a program in the ready state places that program in a paused-ready state.

The following events continue even while a program is paused:

- current and pending motions
- condition handler scanning
- current output pulses
- current and pending system calls
- other asynchronous statements (DELAY, WAIT, etc.)

Even though condition handlers continue to be scanned while the program is paused, the actions are not performed until the program is unpaused.

The PAUSE statement is permitted as a condition handler action. Refer to [Condition Handlers](#) chapter for further information.

Examples:

```
PAUSE
```

```
PAUSE weld_main, weld_util, weld_ctrl
```

```
PAUSE ALL
```

See also:

[UNPAUSE Statement](#)

10.35 PROGRAM Statement

The PROGRAM statement identifies the program and specifies its attributes. It is always the first statement in a PDL2 program.

Syntax:

```
PROGRAM name <attribute_list>
```

Comments:

name is a user-defined identifier used to name the program and the file in which

the program is stored.

The executable section of a program is marked with a BEGIN statement and an END statement.

attribute_list is an optional list, separated by commas, of any of the following attributes:

Default Arm

Syntax:

```
PROG_ARM = int_value
```

Values from 1 to 4 can be used to indicate a default arm for the program. If a default arm is not specified, the value of \$DFT_ARM is used.

Arm State

Syntax:

```
ATTACH | DETACH
```

ATTACH indicates the default arm is to be attached when the program begins execution and DETACH indicates the default arm is to be detached when the program begins execution. The default is ATTACH for holdable programs.

Priority

Syntax:

```
PRIORITY = int_value
```

Values from 1 to 3, with 3 being the highest priority, can be used to indicate a priority for the program (as explained in the "Execution Control" chapter). The default priority is 2.

Classification

Syntax:

```
HOLD | NOHOLD
```

HOLD indicates the program is holdable; NOHOLD indicates the program is non-holdable. The default is HOLD

Stack Size

Syntax:

```
STACK = int_value
```

Values from 0 to 65534 can be used to indicate the stack size. The default value is 1000 bytes.

Issuing the PROGRAM VIEW with the FULL option, the maximum amount of stack used during the program life is displayed. This information can be used for well dimensioning the program stack before creating a final version of the program.

Examples:

```
PROGRAM example1 PROG_ARM = 2, HOLD
-- VAR and CONST section
BEGIN
.
.
.
END example1
```

```
PROGRAM example2 PRIORITY = 3, NOHOLD
```

```
PROGRAM example3
```

See also:

[BEGIN Statement](#)
[END Statement](#)

10.36 PULSE Statement

The PULSE statement reverses the current state of a digital output signal for a specified period of time.

Syntax:

```
PULSE || $DOUT | $BIT || [indx] FOR p_time <ADVANCE>
```

Comments:

indx is an INTEGER expression representing an output port array index.

p_time is an INTEGER expression representing time in milliseconds.

If the optional ADVANCE clause is used, then the following PDL2 statements will execute simultaneously with the PULSE; otherwise, the program is suspended until the PULSE has finished.

If the ADVANCE option is specified and another PULSE statement is executed before the first one is complete, they will overlap. Overlapped pulses cause the time for an existing pulse to be extended, if necessary, to include the time of a new pulse on the same port.

PULSE is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

The PULSE statement is permitted as a condition handler action. However, the ADVANCE option must be used. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

```
PULSE $DOUT[i] FOR 300
```

```
PULSE $DOUT[24] FOR 700 ADVANCE
```

```
PULSE $BIT[4] FOR 200
```

See also:

[Chap.5. - Ports](#)

10.37 PURGE CONDITION Statement

The PURGE CONDITION statement deletes a condition handler definition.

Syntax:

```
PURGE CONDITION||ALL|[int_expr]<, CONDITION[int_expr]>...||
```

Comments:

int_expr is the number of the condition handler to be purged.

Purged condition handlers cannot be re-enabled.

Condition handler definitions are automatically purged when the program is deactivated.

If the ALL option is specified, all condition handlers defined by the program are

purged.

If a condition handler is currently enabled as part of a WITH clause, it cannot be purged.

It is an error if the program attempts to purge a condition handler that is currently attached in another program.

The PURGE CONDITION statement is permitted as a condition handler action. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

```
PURGE CONDITION[1]
PURGE CONDITION[error_chk], CONDITION[signal_chk]
PURGE ALL
```

See also:

[ATTACH Statement](#)
[CONDITION Statement](#)
[ENABLE CONDITION Statement](#)
[DISABLE CONDITION Statement](#)
[Chap.8. - Condition Handlers](#)

10.38 READ Statement

The READ statement reads input data into program variables from the specified LUN.

The syntax of the READ statement is as follows:

```
READ <lun_var> (var_id <, var_id>...)
```

The *lun_var* can be a variable representing any open LUN or either of the predefined LUNs, LUN_TP or LUN_CRT. If a *lun_var* is not specified, the default LUN indicated by the predefined variable \$DFT_LUN is used.

The *var_id* can be any variable identifier of the following data types:

INTEGER	VECTOR
REAL	POSITION
BOOLEAN	JOINTPOS
STRING	XTNDPOS

However, the *var_id* cannot be a predefined variable which requires limit checking. [Chap.12. - Predefined Variables List](#) describes each predefined variable including whether or not it requires limit checking.

The reserved word NL (new line) can also be used in the list of identifiers to force the next item to be read from a new line.

Each data item that is read is assigned to the corresponding variable from the list of identifiers. Data items can be read in ASCII or binary format, depending on how the LUN has been opened.

Examples of the READ statement follow:

```
READ (body_type, total_units, operator_id)
-- reads three values from the default lun
```

In this case, if the operator entered the values 4, 50, and JOE, the READ statement

would make the following assignments:

```
body_type := 4
total_units := 50
operator_id := JOE
```

The following examples indicate how to read data from a window and a file:

```
OPEN FILE crt2_lun ('CRT2:', 'RW') -- opens window CRT2:
READ crt2_lun (menu_choice) -- reads a value from the CRT2:
window
...
CLOSE FILE crt2_lun

OPEN FILE file_lun ('specs.dat', 'R')
READ file_lun (body_type, NL, total_units, NL, operator_id, NL)
-- reads three values from the file UD:specs.dat
-- expects each value to be at the beginning of a new line
...
CLOSE FILE file_lun
```

See also:

[DV_CNTRL Built-In Procedure](#)
[DECODE Statement](#)
[OPEN FILE Statement](#)
[CLOSE FILE Statement](#)
[WRITE Statement](#)
[Chap.9. - Communication](#)

10.38.1 Format Specifiers

Optional format specifiers can be used to format input. For binary data, a single format specifier can be used to indicate the number of bytes a value occupies. The effect of a format specifier on ASCII data depends on the data type being read.

The syntax of a format specifier is as follows:

```
:: intSpecifier -- integer expression
```

On some data types, a second specifier also is allowed. It follows the first format specifier using the same syntax.

The effects of format specifiers for each data type for ASCII data are as follows:

INTEGER: The first specifier is the maximum number of characters to be read. The second specifier indicates the base of the number. Valid base values are as follows:

- 1 for octal;
- 2 for hexadecimal;
- 3 for character;
- 4 for binary (NOTE: not supported for READ Statement);
- 5 for decimal.

REAL: The format specifier indicates the maximum number of characters to be read. Only one specifier is allowed.

BOOLEAN: The data must be one of the Boolean predefined constants (TRUE, ON,

FALSE, OFF). The format specifier indicates the maximum number of characters to be read. Only one specifier is allowed.

STRING: The format specifier indicates the maximum number of characters to be read. Only one specifier is allowed.

VECTOR, POSITION, JOINTPOS, and XTNDPOS: The formats in which data must be entered for each data type are as follows:

- in all cases, the left angle bracket (<) starts the value and the right angle bracket (>) ends the value. The commas in each value are required;
- for vectors and positions, x, y, and z represent Cartesian location components.

VECTOR: <x, y, z>

For positions, e1, e2, and e3 represent Euler angle components and cnfg_str represents a configuration string. The configuration string is not enclosed in quotes.

POSITION: <x, y, z, e1, e2, e3, cnfg_str>

For jointpos, components that have no meaning with the current arm are left blank, but commas must be used to mark the place. The arm number 'n' for jointpos and xtndpos is preceded by the character 'A'.

JOINTPOS: <j1, j2, j3, j4, j5, j6, An>

XTNDPOS: <<x, y, z, e1, e2, e3, cnfg_str> <x1, ...> An>

The format specifier indicates the maximum number of characters to be read for each component of the item. Only one specifier is allowed.

10.38.2 Power Failure Recovery

A READ statement which is pending on the Teach Pendant or a serial communication line will return an error after power failure recovery has completed.

10.39 RELAX HAND Statement

The RELAX HAND statement relaxes a hand (tool).

Syntax:

```
RELAX HAND number <FOR || ARM[n] <,ARM[n]>... | ALL || >
```

Comments:

number indicates the number of the hand to be relaxed. Two hands are available per arm.

The effect of the relax operation depends on the type of hand being operated (refer to [Chap.4. - Motion Control](#)) configured using the HAND configuration tool delivered with the system software.

The optional FOR ARM clause can be used to indicate a particular arm, a list of arms, or all arms. If not specified, the default arm is used.

Examples:

```
RELAX HAND 1
```

```
RELAX HAND 2 FOR ARM[2]
```

See also:

[CLOSE HAND Statement](#)
[OPEN HAND Statement](#)

[Chap.4. - Motion Control](#)

10.40 REPEAT Statement

The REPEAT statement executes a sequence of statements until a Boolean expression is TRUE.

Syntax:

```
REPEAT
  <statement>...
  UNTIL bool_expr
```

Comments:

bool_expr is any expression that yields a Boolean result.

Since the *bool_expr* is evaluated at the end of the loop, the loop is always executed at least one time, even if *bool_expr* is TRUE when the loop is first encountered.

If the *bool_expr* is FALSE, the loop executes again. If it is TRUE, then the looping stops and the program continues with the statements after the UNTIL.

Examples:

```
REPEAT
  WRITE ('Exiting program',NL)
  WRITE ('Are you sure? (Y/N) : ')
  READ (ans, NL)
  UNTIL (ans = 'Y') OR (ans = 'N')
```

See also:

[WHILE Statement](#)
[FOR Statement](#)

10.41 RESUME Statement

The RESUME statement resumes pending motion resulting from a LOCK Statement.

Syntax:

```
RESUME < | | ARM[n] <,ARM[n]>... | ALL | | >
```

Comments:

The programmer can specify motion on an arm, a list of arms, or all arms is to be resumed. If nothing is specified, motion on the default arm is resumed.

The arm must be unlocked using an UNLOCK statement before the motion is resumed. An error is detected if a program tries to RESUME motion for an arm that is still locked or attached by another program.

The RESUME statement is permitted as a condition handler action. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

```
RESUME
RESUME ARM[3], ARM[1]
RESUME ALL
```

See also:

[LOCK Statement](#)
[UNLOCK Statement](#)

10.42 RETURN Statement

The RETURN statement returns program control from the routine currently being executed to the place from which the routine was called.

Syntax:

```
RETURN <(result)>
```

Comments:

(*result*) is required when the RETURN statement is in a function routine. *result* is an expression of the same type as the function routine.

The RETURN statement is required for function routines. If no RETURN is executed before the END of that routine is reached, an error will occur.

In a procedure routine, the RETURN is optional. If it is used, however, a result cannot be returned.

Several RETURNS can be used in a routine, but only one will be executed.

Examples:

```
ROUTINE time_out : BOOLEAN
-- checks to see if input is received within time limit
CONST
    time_limit = 3000
VAR
    time_slice : INTEGER
BEGIN
    $TIMER[1] := 0
REPEAT
    time_slice := $TIMER[1]
UNTIL ($DIN[1] = ON) OR (time_slice > time_limit)
IF time_slice > time_limit THEN
    RETURN (TRUE)
ELSE
    RETURN (FALSE)
ENDIF
END time_out
```

See also:

[ROUTINE Statement](#)
[Chap.7. - Routines](#)

10.43 ROUTINE Statement

The ROUTINE statement declares a user-defined routine in the declaration section of a program.

Syntax:

Procedure Routine:

```
ROUTINE proc_name <param_list>
    <constant and variable declarations>
BEGIN
    <statement...>
END proc_name
```

Function Routine:

```
ROUTINE func_name <param_list> : return_type
    <constant and variable declarations>
```

```

BEGIN
  <statement...> -- must contain a RETURN statement
END func_name
Param_list:
  <(id <, id>... : id_type <; id <, id>... : id_type>...)>
Shared Routine:
  ROUTINE name <param_list> <: return_type> EXPORTED FROM
  prog_name

```

Comments:

proc_name or *func_name* are user-defined identifiers that specify the name of the routine.

VAR and CONST declarations are used to declare variables and constants local to the routine.

return_type is the data type of the value returned by a function.

An optional *param_list* can be used to specify what data items are passed to the routine. *id* is a user-defined identifier, and *id_type* is the data type of the parameter. Routines can be declared to be owned by another program or to be public for use by other programs using the optional EXPORTED FROM clause.

For shared routines, *prog_name* indicates the name of the program owning the routine. The declaration and executable sections of the routine appear only in the program that owns the routine.

Refer to [Chap.7. - Routines](#) for more information about the declaration and usage of routines.

Examples:

```

ROUTINE positive (x : INTEGER) : BOOLEAN
BEGIN
  RETURN (x >= 0)
END positive

ROUTINE push (stack : stack_type; item : POSITION)
BEGIN
  stack.top := stack.top + 1
  stack.data[stack.top] := item
END push

```

See also:

[RETURN Statement](#)
[Chap.7. - Routines](#)

10.44 SELECT Statement

The SELECT statement is used to choose between several alternative courses of action, based on the result of an INTEGER expression.

Syntax:

```

SELECT int_expr OF
  CASE (int_val <, int_val>...) :
    <statement>...
  <CASE (int_val <, int_val>...) :
    <statement>... >...
<ELSE:

```

```

<statement>... >
ENDSELECT

```

Comments:

The SELECT statement tries to match the value of *int_expr* with an *int_val*. If a match is found the statement(s) following that CASE are executed and the rest of the cases are skipped.

The optional ELSE clause, if used, will be executed if no match is found. If there is no match and the ELSE clause is not used, an error will result.

int_val is a literal, a predefined constant, or a user-defined INTEGER constant. No two CASE clauses can use the same INTEGER value.

Examples:

```

SELECT tool_type OF
CASE (1):
    $TOOL := utool_weld
        style_weld
CASE (2):
    $TOOL := utool_grip
        style_grip
CASE (3)
    $TOOL := utool_paint
        style_paint
ELSE:
    tool_error(tool_type)
ENDSELECT

```

See also:

[IF Statement](#)

10.45 SIGNAL Statement

The SIGNAL statement is used when the use of a limited resource is finished (when applied to a SEMAPHORE or a PATH) or for triggering user events conditions.

Syntax:

SIGNAL || *semaphore_var* | SEGMENT *path_var* | EVENT *user_event_code* | |

Comments:

The SIGNAL statement indicates the specified resource is available for use by other programs that might be waiting for it.

semaphore_var is a SEMAPHORE variable. *semaphore_var* must be initialized with at least one SIGNAL or there will be a deadlock.

If programs are waiting on the *semaphore_var* when the SIGNAL statement is executed, the first waiting program will be resumed. If no programs are waiting, the signal is remembered so that the next program to WAIT on the *semaphore_var* will not actually wait.

The SIGNAL SEGMENT statement resumes path motion that is currently suspended. Path motion will be suspended if the \$SEG_WAIT field of a node is TRUE. The only way to resume the path motion is to execute a SIGNAL SEGMENT statement.

The \$SEG_WAIT field is a BOOLEAN indicating whether or not processing of the path should be suspended until the path is signalled. This field is used to obtain synchronization between path segment processing and other aspects of the

application such as sensor detection, mutual exclusion, etc. If the value of the \$SEG_WAIT field is FALSE, path processing is not suspended at that node.

If the SIGNAL SEGMENT statement is executed and the path specified is not currently suspended, the statement will have no effect.

The SIGNAL EVENT statement satisfies a specific user event (see [Chap.8. - Condition Handlers](#)) if any condition with the same event number is defined and enabled in the system. If no conditions with such event number are enabled, the statement will have no effect. Possible numbers allowed for user events are included in the range 49152-50175.

The SIGNAL statement is permitted as a condition handler action. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

```
SIGNAL resource
SIGNAL SEGMENT weld_path
SIGNAL sem[1]
SIGNAL EVENT 50100.
```

See also:

[WAIT Statement](#)
[Chap.4. - Motion Control](#)

10.46 TYPE Statement

The TYPE statement marks the beginning of a type declaration section in a program.

Syntax:

```
TYPE
  type_name = RECORD
    name <, name>... : Data_type
    <name <, name>... : data_type>...
  ENDRECORD

  type_name = NODEDEF
    <predefined_name <, predefined_name>... <NOTEACH> >...
    <name <, name>... : data_type <NOTEACH> >...
  ENDNODEDEF
```

Comments:

A type declaration establishes a new user-defined data type that can be used when declaring variables and parameters.

type_name can be any user-defined identifier.

User-defined data types are available to the whole system. Make sure that unique names are used to avoid conflicts.

Field *names* are local to the user-defined data type. This means two different user-defined types can contain a field having the same *name*.

Valid field data types are explained in [Chap.3. - Data Representation](#).

The TYPE statement is not allowed in ROUTINES.

The TYPE statement must come before any variable declaration that uses user-defined fields.

A NODEDEF type defines a structure including both *predefined_name* fields and user defined fields. Note that it could be very useful to define the structure of a

PATH NODE.

A NODEDEF type can contain any number of *predefined_name* fields and any number of *name* fields. However, the NODEDEF must contain at least one field.

predefined_name is a standard node field having the same meaning as the corresponding predefined variable. For example, \$MOVE_TYPE can be used and has the same meaning as described in [Chap.12. - Predefined Variables List](#).

The NOTEACH option in a NODEDEF type indicates that those fields are not displayed in the teach environment. This disables the user from modifying those fields while teaching.

Refer to [Chap.3. - Data Representation](#) for a list of valid *predefined_name* fields and further information about RECORD and NODEDEF type definitions.

Examples:

```

PROGRAM main
TYPE
    ddd_part = RECORD
        name: STRING[15]
        count: INTEGER
        params: ARRAY[5] OF REAL
    ENDRECORD

    lapm_pth1 = NODEDEF
        $MAIN_POS
        $MOVE_TYPE
        $ORNT_TYPE
        $SPD_OPT
        $SEG_TERM_TYPE
        $SING_CARE
        weld_sch : ARRAY[8] OF REAL
        gun_on : BOOLEAN
    ENDNODEDEF

VAR
    count, index : INTEGER
    part_rec : ddd_part
    weld_pth : PATH OF lapm_pth1

BEGIN
    -- Executable section
END main

```

See also:

[VAR Statement](#)
[Chap.3. - Data Representation](#)

10.47 UNLOCK Statement

The UNLOCK statement allows motion to be restarted on a locked arm. The motion is not resumed until a RESUME statement is executed.

Syntax:

UNLOCK < || ARM[n] <, ARM[n]>... | ALL || >

Comments:

The UNLOCK statement must be issued from the same program that issued the LOCK statement.

The programmer can specify an arm, a list of arms, or all arms are to be unlocked. If nothing is specified, the default arm is unlocked.

To resume pending motion, a RESUME statement must be issued after the UNLOCK statement.

The UNLOCK statement is permitted as a condition handler action. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

UNLOCK

UNLOCK ARM [4] , ARM [5]

UNLOCK ALL

See also:

[LOCK Statement](#)

[RESUME Statement](#)

10.48 UNPAUSE Statement

The UNPAUSE statement unpauses paused programs. The effect of unpausing a program depends on the holdable/non-holdable program attribute.

Syntax:

UNPAUSE || *prog_name* <, *prog_name*>... | ALL ||

Comments:

If *prog_name* or a list of names is specified, those programs are unpause. If no name is specified, the program issuing the statement is unpause. If ALL is specified, all paused programs are unpause.

If the statement is issued from a holdable program, the programs are placed in the running state. If the statement is issued from a non-holdable program, holdable programs are placed in the ready state and non-holdable programs are placed in the running state.

The statement has no effect on programs that are not paused.

The UNPAUSE statement is permitted as a condition handler action. Refer to [Chap.8. - Condition Handlers](#) for further information.

Examples:

UNPAUSE *weld_main*, *weld_util*, *weld_cntrl*

UNPAUSE ALL

See also:

[PAUSE Statement](#)

10.49 VAR Statement

The VAR statement marks the beginning of a variable declaration section in a program or routine.

Syntax:

VAR :

```

        name <, name>... : data_type <var_options>
        <name <, name>... : data_type <var_options>>...
    
```

Shared Variables:

```

        name<, name>...:data_type      EXPORTED      FROM      prog_name
        <var_options>
    
```

Var_Options:

```

        <( initial_value )> <NOSAVE> <NODATA>
    
```

Comments:

A variable declaration establishes a variable identifier with a name and a data type. *name* can be any user-defined identifier not previously used in the same scope.

This means a program cannot have two variables of the same *name*.

Valid data types are explained in [Chap.3. - Data Representation](#).

The variable declaration section is located between the PROGRAM or ROUTINE statement and the BEGIN statement.

Variables can be declared to be owned by another program or to be public for use by other programs using the optional EXPORTED FROM clause. For shared variables, *prog_name* indicates the name of the program owning the variable.

If the NOSAVE clause is specified, the variables included in that declaration are not saved to the variable file (.VAR file).

The NOSAVE and EXPORTED FROM clauses are not permitted on routine VAR declarations.

The NODATA attribute allows to avoid displaying the corresponding variable on the TP DATA Page (see also [par. 16.2 User table creation from DATA environment on page 558](#)).

The *initial_value* option is permitted on REAL, INTEGER, BOOLEAN, and STRING declarations. It is used to indicate a value to be assigned to the variable before the BEGIN of the program or routine is executed.

Examples:

```

PROGRAM main
VAR
    count, index : INTEGER (0) NOSAVE
    angle, dist : REAL
    job_complete : BOOLEAN EXPORTED FROM main
    error_msg : STRING[30] EXPORTED FROM error_chk
    menu_choices : ARRAY[4] OF STRING[30]
    matrix : ARRAY[2,10] OF INTEGER
    offset : VECTOR
    pickup, perch : POSITION EXPORTED FROM data1
    option : STRING[10] (backup) NOSAVE
    safety_pos : JOINTPOS FOR ARM[2]
    door_frame : XTNDPOS FOR ARM[3]
    work_area : SEMAPHORE NOSAVE
    default_part : INTEGER (0xFF) NOSAVE
BEGIN
    -- Executable section
END main
    
```

See also:

[CONST Statement](#)
[TYPE Statement](#)

Chap.3. - Data Representation

10.50 WAIT Statement

The WAIT statement requests access to a limited resource.

Syntax:

```
WAIT semaphore_var
```

Comments:

semaphore_var is a SEMAPHORE variable.

If the requested resource is not available for use, the program will wait until it becomes available. A resource becomes available when a SIGNAL statement is executed.

WAIT is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

Examples:

```
WAIT resource
WAIT sem[1]
```

See also:

[SIGNAL Statement](#)

10.51 WAIT FOR Statement

The WAIT FOR statement suspends program execution until the specified condition is met.

Syntax:

```
WAIT FOR cond_expr
```

Comments:

cond_expr specifies a list of conditions for which the program will wait. The expression can be constructed with AND and OR operators.

Conditions are described in [Chap.8. - Condition Handlers](#).

When the *cond_expr* is satisfied, program execution continues.

WAIT FOR is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes.

It is possible to disable the monitoring of the WAIT FOR condition during the entire HELD state of the program. This is accomplished by setting bit 5 of \$PROG_CNFG equal to 1. If it is desired to have all programs operate in this manner, it is sufficient to set bit 5 of \$CNTRL_CNFG to 1 at system startup. (This bit is copied into bit 5 of \$PROG_CNFG at every program activation).

Disabling the monitoring of the WAIT FOR as described above will apply to all kinds of expressions, states, and events present in a WAIT FOR. Note that bit 5 of \$PROG_CNFG is considered at the moment of the WAIT FOR interpretation so that it is possible to alter it inside the interrupt service routine. This is not recommended however, as the state of the bit could be unclear in other areas of the program.

Examples:

```
WAIT FOR $DOUT[24] +
```

See also:

[DELAY Statement](#)
[Chap.8. - Condition Handlers](#)

10.52 WHILE Statement

The WHILE statement executes a sequence of statements as long as a Boolean expression is TRUE.

Syntax:

```
WHILE bool_expr DO
    <statement>...
ENDWHILE
```

Comments:

bool_expr is any expression that yields a Boolean result.
 If *bool_expr* is initially FALSE, the loop will never execute; it will be skipped entirely.
 If *bool_expr* is TRUE, the loop will execute and then test *bool_expr* again.

Examples:

```
WHILE num < num_errors DO
    IF priority_index[num] < 2 THEN
        WRITE(err_text[num], ` (non critical)`, NL)
    ELSE
        WRITE(err_text[num], ` ***CRITICAL***`, NL)
    ENDIF
    num := num + 1
ENDWHILE
```

See also:

[FOR Statement](#)
[REPEAT Statement](#)

10.53 WRITE Statement

The WRITE statement writes output data from a program to the specified LUN.

The syntax of the WRITE statement is as follows:

```
WRITE <lun_var> (expr <, expr>...)
```

The *lun_var* can be a variable representing any open LUN or either of the predefined LUNs, LUN_TP, LUN_CRT or LUN_NULL. If a *lun_var* is not specified, the default LUN indicated by the predefined variable \$DFT_LUN is used.

The *expr* can be any expression of the following data types:

INTEGER	VECTOR
REAL	POSITION
BOOLEAN	JOINTPOS
STRING	XTNDPOS

The reserved word NL (new line) can also be used as a data item. NL causes a new line to be output.

Each data item is written out in the order that it is listed. Data items can be written in ASCII or binary format, depending on how the LUN has been opened.

For example, the following statement writes out a string literal followed by a new line to the default output device:

```
WRITE ('Enter body style, units to process, and operator id.', NL)
```

Notice the string literal is enclosed in single quotes ('), which are not written as part of the output.

The WRITE statement is executed asynchronously when writing to certain types of devices such as communication devices. This is done so other programs can execute simultaneously without having to wait for the statement to be completed. The predefined variable \$WRITE_TOUT indicates a timeout period for asynchronous writes.

10.53.1 Output Buffer Flushing

Internally, write requests are stored in an output buffer until the buffer is full or some other event causes the buffer to be flushed. The output buffer is flushed by the end of a WRITE statement, by the reserved word NL or the ASCII newline code, or by a CLOSE FILE statement.

10.53.2 Format Specifiers

As with the READ statement, optional format specifiers can be used in formatting output. For binary data, a single format specifier can be used to indicate the number of bytes a value occupies. The effect of a format specifier on ASCII data depends on the data type being written.

The syntax of a format specifier is as follows:

```
:: intSpecifier -- integer expression
```

On some data types, a second specifier also is allowed. It follows the first format specifier using the same syntax.

The effects of format specifiers for each data type for ASCII data are as follows:

INTEGER: The first specifier is the minimum number of characters to be written. If the value requires more characters, as many as are needed will be used. The specifier can be a positive or negative value. A positive value means right-justify the number and is the default if no sign is used. A negative value means left-justify the number.

The second specifier indicates the base of the number. Valid base values are as follows:

- 1 for octal;
- 2 for hexadecimal;
- 3 for character;
- 4 for binary;
- 5 for decimal.

If the second write format specifier is negative then the value is preceded by zeros instead of blanks. For example:

```
WRITE (1234::6::-2) -- Is written as 0004D2 (Hex)
```

REAL: The first specifier is the minimum number of characters to be written. If the value requires more characters, as many as are needed will be used. The specifier can be a positive or negative value. A positive value means right-justify the number and is the default if no sign is used. A negative value means left-justify the number.

The second specifier, if positive, is the number of decimal places to use. A negative number specifies scientific notation is to be used.

BOOLEAN: The value written will be one of the Boolean predefined constants (TRUE or FALSE). The format specifier indicates the number of characters to be written. The specifier can be a positive or negative value. A positive value means right-justify the item and is the default if no sign is used. A negative value means left-justify the item. Only one specifier is allowed.

STRING: The format specifier indicates the number of characters to be written. The specifier can be a positive or negative value. A positive value means right-justify the string and is the default if no sign is used. A negative value means left-justify the string. Only one specifier is allowed. Note that the quotes are not written.

If a format specifier greater than 143 characters is specified, the write operation will fail. To avoid this, do not use a format specifier for large output strings.

VECTOR, POSITION, JOINTPOS, and XTNDPOS: The formats in which data is output for each data type are as follows:

- in all cases, the left angle bracket (<) starts the value, the right angle bracket (>) ends the value, and commas separate each component;
- for vectors and positions, x, y, and z represent Cartesian location components.

VECTOR: <x, y, z>

For positions, e1, e2, and e3 represent Euler angle components and cnfg_str represents a configuration string. The configuration string is not enclosed in quotes.

POSITION: <x, y, z, e1, e2, e3, cnfg_str>

For jointpos, components that have no meaning with the current arm are left blank, but commas are used to mark the place. The arm number 'n' for jointpos and xtndpos is preceded by the character 'A'.

JOINTPOS: <j1, j2, j3, j4, j5, j6, An>

XTNDPOS: <<x, y, z, e1, e2, e3, cnfg_str> <x1, ...> An>

The format specifier indicates the maximum number of characters to be written for each component of the item.

The second specifier, if positive, is the number of decimal places to use. A negative number specifies scientific notation is to be used.

10.53.3 Power Failure Recovery

A WRITE statement to a serial communication line will return an error after power failure recovery. A WRITE statement to the teach pendant will not return an error, and the windows will contain the correct display data after the system is restarted.

Examples:

```

        WRITE (x, y, z) -- lun_var defaults to $DFT_LUN
        WRITE LUN_TP ('The value of x is ', x)
        WRITE LUN_NULL ('This string will disappear')
        OPEN FILE crt1_lun ('CRT1:', 'RW')
        WRITE crt1_lun (num, NL, error_msg)
    
```

See also:

[DV_CNTRL Built-In Procedure](#)

[CLOSE FILE Statement](#)

[OPEN FILE Statement](#)

READ Statement
ENCODE Statement
Chap.9. - Communication

11. BUILT-IN ROUTINES LIST

This chapter is an alphabetical reference of PDL2 built-in routines. The following information is provided for each built-in:

- short description
- calling sequence
- parameter description
- comments concerning usage
- example

This chapter uses the syntax notation explained in [Introduction to PDL2](#) chapter to represent PDL2 built-in routines.

In the “comments” area of each built-in function description, references to parameters are italicized to indicate the argument name.

The following groups of built-in routines are listed below:

- [Math Built-In Routines](#)
- [Arm Built-In Routines](#)
- [Serial Input/Output Built-In Routines](#)
- [Path Built-In Routines](#)
- [Position Built-In Routines](#)
- [Screen Built-In Routines](#)
- [Window Built-In Routines](#)
- [String Built-In Routines](#)
- [Bit Manipulation Built-In Routines](#)
- [System Data Built-In Routines](#)
- [Error Handling Built-In Routines](#)
- [Misc Built-In Routines](#)

Following is a list of all built-in routines and procedures belonging to the listed above groups.



As far as concerns VP2 built-in routines, please refer to [VP2 - Visual PDL2 Manual, chapter 6](#).

Math Built-In Routines

[ABS Built-In Function](#)
[ACOS Built-In Function](#)
[ASIN Built-In Function](#)
[ATAN2 Built-In Function](#)
[COS Built-In Function](#)
[EXP Built-In Function](#)
[LN Built-In Function](#)
[RANDOM Built-in Function](#)

	ROUND Built-In Function SIN Built-In Function SQRT Built-In Function TAN Built-In Function TRUNC Built-In Function
Arm Built-In Routines	ARM_COLL_THRS Built-In Procedure ARM_COOP Built-In Procedure ARM_GET_NODE Built-In Function ARM_JNTP Built-In Function ARM_MCOOP Built-In Procedure ARM_MOVE_ATVEL Built-in Procedure ARM_NUM Built-In Function ARM_PALLETIZING Built-In Procedure ARM_POS Built-In Function ARM_SET_NODE Built-In Procedure ARM_SOFT Built-In Procedure ARM_XTND Built-In Function AUX_COOP Built-In Procedure AUX_DRIVES Built-In Procedure AUX_SET Built-In Procedure CONV_TRACK Built-In Procedure HDIN_READ Built-In Procedure HDIN_SET Built-In Procedure JNT_SET_TAR Built-In Procedure ON_JNT_SET Built-In Procedure ON_JNT_SET_DIG Built-In Procedure ON_POS Built-In Procedure ON_POS_SET Built-In Procedure ON_POS_SET_DIG Built-In Procedure ON_TRAJ_SET Built-In Procedure ON_TRAJ_SET_DIG Built-In Procedure SENSOR_GET_DATA Built-In Procedure SENSOR_GET_OFST Built-In Procedure SENSOR_SET_DATA Built-In Procedure SENSOR_SET_OFST Built-In Procedure SENSOR_TRK Built-In Procedure STANDBY Built-In Procedure
Serial Input/Output Built-In Routines	DIR_GET Built-In Function DIR_SET Built-In Procedure DV_CCTRL Built-In Procedure DV_STATE Built-In Function EOF Built-In Function FL_BYTES_LEFT Built-In Function FL_GET_POS Built-In Function FL_SET_POS Built-In Procedure FL_STATE Built-In Function VOL_SPACE Built-In Procedure
Path Built-In Routines	NODE_APP Built-In Procedure NODE_DEL Built-In Procedure NODE_GET_NAME Built-In Procedure NODE_INS Built-In Procedure NODE_SET_NAME Built-In Procedure PATH_GET_NODE Built-In Procedure PATH_LEN Built-In Function

Position Built-In Routines

[JNT](#) Built-In Procedure
[JNTP_TO_POS](#) Built-In Procedure
[POS](#) Built-In Function
[POS_COMP_IDL](#) Built-In Procedure
[POS_CORRECTION](#) Built-In Procedure
[POS_FRAME](#) Built-In Function
[POS_GET_APPR](#) Built-In Function
[POS_GET_CNFG](#) Built-In Function
[POS_GET_LOC](#) Built-In Function
[POS_GET_NORM](#) Built-In Function
[POS_GET_ORNT](#) Built-In Function
[POS_GET_RPY](#) Built-In Procedure
[POS_IDL_COMP](#) Built-In Procedure
[POS_IN_RANGE](#) Built-In Procedure
[POS_INV](#) Built-In Function
[POS_MIR](#) Built-In Function
[POS_SET_APPR](#) Built-In Procedure
[POS_SET_CNFG](#) Built-In Procedure
[POS_SET_LOC](#) Built-In Procedure
[POS_SET_NORM](#) Built-In Procedure
[POS_SET_ORNT](#) Built-In Procedure
[POS_SET_RPY](#) Built-In Procedure
[POS_SHIFT](#) Built-In Procedure
[POS_TO_JNTP](#) Built-In Procedure
[POS_XTRT](#) Built-In Procedure
[VEC](#) Built-In Function

Screen Built-In Routines

[SCRN_ADD](#) Built-In Procedure
[SCRN_CLEAR](#) Built-In Procedure
[SCRN_CREATE](#) Built-In Function
[SCRN_DEL](#) Built-In Procedure
[SCRN_GET](#) Built-In Function
[SCRN_REMOVE](#) Built-In Procedure
[SCRN_SET](#) Built-In Procedure

Window Built-In Routines

[WIN_ATTR](#) Built-In Procedure
[WIN_CLEAR](#) Built-In Procedure
[WIN_COLOR](#) Built-In Procedure
[WIN_CREATE](#) Built-In Procedure
[WIN_DEL](#) Built-In Procedure
[WIN_DISPLAY](#) Built-In Procedure
[WIN_GET_CRSR](#) Built-In Procedure
[WIN_LINE](#) Built-In Function
[WIN_LOAD](#) Built-In Procedure
[WIN_POPUP](#) Built-In Procedure
[WIN_REMOVE](#) Built-In Procedure
[WIN_SAVE](#) Built-In Procedure
[WIN_SEL](#) Built-In Procedure
[WIN_SET_CRSR](#) Built-In Procedure
[WIN_SIZE](#) Built-In Procedure
[WIN_STATE](#) Built-In Function

String Built-In Routines

[CHR](#) Built-In Procedure
[ORD](#) Built-In Function
[STR_CAT](#) Built-In Function
[STR_CODING](#) Built-In Function
[STR_CONVERT](#) Built-In Function
[STR_DEL](#) Built-In Procedure
[STR_EDIT](#) Built-In Procedure

	STR_GET_INT Built-In Function STR_GET_REAL Built-In Function STR_INS Built-In Procedure STR_LEN Built-In Function STR_LOC Built-In Function STR_OVS Built-In Procedure STR_SET_INT Built-In Procedure STR_SET_REAL Built-In Procedure STR_TO_IP Built-In Function STR_TOKENS Built-In Function STR_XTRT Built-In Procedure
Bit Manipulation Built-In Routines	BIT_ASSIGN Built-In Procedure BIT_CLEAR Built-In Procedure BIT_FLIP Built-In Function BIT_SET Built-In Procedure BIT_TEST Built-In Function
System Data Built-In Routines	CLOCK Built-In Function DATE Built-In Function KEY_LOCK Built-In Procedure MEM_SPACE Built-In Procedure SYS_CALL Built-In Procedure
Error Handling Built-In Routines	ACT_POST Built-in Procedure ERR_POST Built-In Procedure ERR_STR Built-In Function ERR_TRAP Built-In Function ERR_TRAP_OFF Built-In Procedure ERR_TRAP_ON Built-In Procedure
Misc Built-In Routines	ARG_COUNT Built-In Function ARG_GET_VALUE Built-in Procedure ARG_INFO Built-In Function ARG_SET_VALUE Built-in Procedure ARRAY_DIM1 Built-In Function ARRAY_DIM2 Built-In Function COND_ENABLED Built-In Function COND_ENBL_ALL Built-In Procedure DRIVEON_DSBL Built-In Procedure FLOW_MOD_ON Built-In Procedure FLOW_MOD_OFF Built-In Procedure IP_TO_STR Built-in Function IR_SET Built-In Procedure IR_SET_DIG Built-In Procedure IR_SWITCH Built-In Procedure IS_FLY Built-In Function PROG_STATE Built-In Function STR_GET_INT Built-In Function STR_GET_REAL Built-In Function STR_SET_INT Built-In Procedure STR_SET_REAL Built-In Procedure

STR_TO_IP Built-In Function
STR_TOKENS Built-In Function
STR_XTRT Built-In Procedure
TABLE_ADD Built-In Procedure
TABLE_DEL Built-In Procedure
TREE_ADD Built-In Procedure
TREE_CLONE Built-In Procedure
TREE_CREATE Built-In Function
TREE_DEL Built-In Procedure
TREE_LOAD Built-In Procedure
TREE_NODE_INFO Built-In Procedure
TREE_NODE_CHILD Built-In Procedure
TREE_REMOVE Built-In Procedure
TREE_SAVE Built-In Procedure
VAR_CLONE Built-in Procedure
VAR_CREATE Built-In Procedure
VAR_INFO Built-In Function
VAR_LINK Built-In Procedure
VAR_LINKS Built-In Procedure
VAR_LINKSS Built-In Procedure
VAR_LINK_INFO Built-In Procedure
VAR_UNLINK Built-In Procedure
VAR_UNLINK_ALL Built-In Procedure
VAR_UNINIT Built-In Function

11.1 ABS Built-In Function

The ABS built-in function returns the absolute value of a specified number.

Calling Sequence:	ABS (number)
Return Type:	REAL INTEGER
Parameters:	number : REAL INTEGER [IN]
Comments:	<p><i>number</i> specifies a positive or negative number. <i>number</i> must be in the normal range for the data type. The return type is the same as the type of <i>number</i>. For example, if <i>number</i> is a REAL then the returned value will be a REAL.</p>
Examples:	<pre>ABS(99.5) -- result is 99.5 ABS(-28.3) -- result is 28.3 ABS(-19) -- result is 19 ABS(324) -- result is 324</pre>

11.2 ACOS Built-In Function

The ACOS built-in function returns the arc cosine of the argument.

Calling Sequence:	ACOS (number)
Return Type:	REAL
Parameters:	number :REAL [IN]
Comments:	<p>The arc cosine is measured in degrees. The result is in the range of 0 to 180 degrees. <i>number</i> specifies a real number in the range of -1 to 1.</p>
Examples:	<pre>ACOS(0.5) -- result is 60 ACOS(-0.5) -- result is 120</pre>

11.3 ACT_POST Built-in Procedure

This built-in posts a message in the user action log file.

Calling Sequence:	ACT_POST (errornum_int, error_string)
Parameters:	errornum_int :INTEGER [IN] error_string :STRING [IN]
Comments:	<p><i>errornum_int</i> is the number of the error to be posted <i>error_string</i> is the message to be posted, associated to <i>errornumber_int</i></p>
Examples:	ACT_POST (43009, 'CEDP action')

11.4 ARG_COUNT Built-In Function

The ARG_COUNT built-in function returns an integer value which is the number of arguments actually passed to the current routine.

Calling Sequence: ARG_COUNT ()

Return Type: INTEGER

Parameters: none

11.5 ARG_GET_VALUE Built-in Procedure

This built-in gets the value of the specified argument.

Calling Sequence: ARG_GET_VALUE (index, variable <, array_index1 <, array_index2>)

Parameters:

index :	INTEGER	[IN]
variable:	ANY TYPE	[IN]
array_index1 :	INTEGER	[IN]
array_index2 :	INTEGER	[IN]

Comments: *index* specifies the argument index in the optional list.
variable indicates the variable in which the required value is copied into.
array_index1 specifies the array row (if the argument is an array)
array_index2 specifies the array column (if the argument is a bidimensional array)

Examples:

```

PROGRAM varargs NOHOLD, STACK = 5000
TYPE aRec = RECORD
    fi : INTEGER
ENDRECORD
TYPE aNode = NODEDEF
    fi : INTEGER
ENDNODEDEF

```

```

VAR
    vi : INTEGER
    vr : REAL (3.1415901) NOSAVE
    vs : STRING[10] ('variable') NOSAVE
    vb : BOOLEAN (TRUE) NOSAVE
    vp : POSITION
    vx : XTNDPOS
    vj : JOINTPOS
    vm : SEMAPHORE NOSAVE
    ve : aRec
    wi : ARRAY[5] OF INTEGER
    wr : ARRAY[5] OF REAL
    ws : ARRAY[5] OF STRING[10]
    wb : ARRAY[5] OF BOOLEAN
    wp : ARRAY[5] OF POSITION
    wx : ARRAY[5] OF XTNDPOS
    wj : ARRAY[5] OF JOINTPOS
    wm : ARRAY[5] OF SEMAPHORE
    we : PATH OF aNode
    vi_value : INTEGER
ROUTINE r2(ai_value : INTEGER; ...) : INTEGER EXPORTED FROM
varargs global
ROUTINE r2(ai_value : INTEGER; ...) : INTEGER
VAR li_dtype, li_array_dim1, li_array_dim2 : INTEGER
    li, lj, lk : INTEGER
    li_value : INTEGER

```

```

lr_value : REAL
ls_value : STRING[20]
lb_value : BOOLEAN
lv_value : VECTOR
lp_value : POSITION
lx_value : XTNDPOS
lj_value : JOINTPOS
mi_value : ARRAY[5] OF INTEGER
mr_value : ARRAY[5] OF REAL
ms_value : ARRAY[5] OF STRING[10]
mb_value : ARRAY[5] OF BOOLEAN
mv_value : ARRAY[5] OF VECTOR
mp_value : ARRAY[5] OF POSITION
mx_value : ARRAY[5] OF XTNDPOS
mj_value : ARRAY[5] OF JOINTPOS
lb_byref : BOOLEAN

BEGIN
  WRITE LUN_CRT ('In r2. Number of arguments', ARG_COUNT, NL)
  FOR li := 1 TO ARG_COUNT DO
    li_dtype := ARG_INFO(li,lb_byref, li_array_dim1,
    li_array_dim2)
    WRITE LUN_CRT ('Index:', li, ' Datatype = ', li_dtype, '[',
    li_array_dim1, ',', li_array_dim2, ']. By ref:', lb_byref, NL)
    SELECT ARG_INFO(li) OF
      CASE (1):
        ARG_GET_VALUE(li, li_value)
        WRITE LUN_CRT ('Int Value = ', li_value, NL)
        ARG_GET_VALUE(li, ai_value)
        WRITE LUN_CRT ('Int Value = ', ai_value, NL)
        ARG_GET_VALUE(li, vi_value)
        WRITE LUN_CRT ('Int Value = ', vi_value, NL)
        li_value += 10
        ARG_SET_VALUE(li, li_value)
      CASE (2): -- Real
        ARG_GET_VALUE(li, lr_value)
        WRITE LUN_CRT ('Rea Value = ', lr_value, NL)
      CASE (3): -- Boolean
        ARG_GET_VALUE(li, lb_value)
        WRITE LUN_CRT ('Boo Value = ', lb_value, NL)
      CASE (4): -- String
        ARG_GET_VALUE(li, ls_value)
        WRITE LUN_CRT ('Str Value = ', ls_value, NL)
      CASE (5): -- Vector
        ARG_GET_VALUE(li, lv_value)
        WRITE LUN_CRT ('Vec Value = ', lv_value, NL)
      CASE (6): -- Position
        ARG_GET_VALUE(li, lp_value)
        WRITE LUN_CRT ('Pos Value = ', lp_value, NL)
        lp_value := POS(0)
        ARG_SET_VALUE(li, lp_value)
      CASE (7): -- Jointpos
        ARG_GET_VALUE(li, lj_value)
        WRITE LUN_CRT ('Jnt Value = ', lj_value, NL)
      CASE (8): -- Xtndpos
        ARG_GET_VALUE(li, lx_value)
        WRITE LUN_CRT ('Xtn Value = ', lx_value, NL)
  
```

```

CASE (31): -- Array of integer
    ARG_GET_VALUE(li, mi_value)
    WRITE LUN_CRT ('Array Int Value = ', mi_value[5], NL)
    IF li_array_dim2 = 0 THEN
        FOR lj := 1 TO li_array_dim1 DO
            ARG_GET_VALUE(li, li_value, lj, 0)
            WRITE LUN_CRT ('Array Int Value[', lj:::-1, ']=',
                           li_value, NL)
        ENDFOR
    ENDIF
CASE (32): -- Array of real
    ARG_GET_VALUE(li, mr_value)
    WRITE LUN_CRT ('Array Rea Value = ', mr_value[5], NL)
CASE (33): -- Array of boolean
    ARG_GET_VALUE(li, mb_value)
    WRITE LUN_CRT ('Array Boo Value = ', mb_value[5], NL)
CASE (34): -- Array of string
    ARG_GET_VALUE(li, ms_value)
    WRITE LUN_CRT ('Array Str Value = ', ms_value[5], NL)
    ENCODE (ls_value, DATE)
    ARG_SET_VALUE(li, ls_value, 2)
CASE (35): -- Array of vector
    ARG_GET_VALUE(li, mv_value)
    WRITE LUN_CRT ('Array Vec Value = ', mv_value[5], NL)
CASE (36): -- Array of position
    ARG_GET_VALUE(li, mp_value)
    WRITE LUN_CRT ('Array Vec Value = ', mp_value[5], NL)
CASE (37): -- Array of jointpos
    ARG_GET_VALUE(li, mj_value)
    WRITE LUN_CRT ('Array Vec Value = ', mj_value[5], NL)
CASE (38): -- Array of xtndpos
    ARG_GET_VALUE(li, mx_value)
    WRITE LUN_CRT ('Array Vec Value = ', mx_value[5], NL)
CASE (0): -- Optional parameters
    WRITE LUN_CRT ('Optional parameter', NL)
ENDSELECT
ENDFOR
RETURN (ARG_COUNT)
END r2

```

See also:

[ARG_COUNT Built-In Function](#)
[ARG_INFO Built-In Function](#)
[ARG_SET_VALUE Built-in Procedure](#)

11.6 ARG_INFO Built-In Function

The ARG_INFO built-in function returns the data type of the specified argument.

Calling Sequence: ARG_INFO (index<, by_reference> <, size_array1> <, size_array2>)

Return Type: INTEGER

Parameters:

index :	INTEGER	[IN]
by_reference:	BOOLEAN	[IN]
size_array1 :	INTEGER	[IN]
size_array2 :	INTEGER	[IN]

Comments:

- *index* specifies the argument index in the parameters list.
- *by_reference* indicates whether the argument is passed by reference or not.
- *size_array1* specifies the array 1st dimension (if the argument is an array).
- *size_array2* specifies the array 2nd dimension (if the argument is a bidimensional array).

11.7 ARG_SET_VALUE Built-in Procedure

This built-in updates the value of the specified argument.

Calling Sequence: `ARG_SET_VALUE (index, variable <, array_index1 <, array_index2>)`

Parameters:

index :	INTEGER	[IN]
variable:	ANY TYPE	[IN]
array_index1 :	INTEGER	[IN]
array_index2 :	INTEGER	[IN]

Comments:

- *index* specifies the argument index in the optional list .
- *variable* indicates the variable containing the value to be set to the specified argument.
- *array_index1* specifies the array row (if the argument is an array)
- *array_index2* specifies the array column (if the argument is a bidimensional array)

Examples: see examples in [ARG_GET_VALUE Built-in Procedure](#) section.

[ARG_COUNT Built-In Function](#)

[ARG_INFO Built-In Function](#)

[ARG_GET_VALUE Built-in Procedure](#)

11.8 ARM_COLL_THRS Built-In Procedure

The ARM_COLL_THRS built-in procedure calculates the Collision Detection sensitivity thresholds. This routine has to be executed during the robot work cycle.

Calling Sequence: `ARM_COLL_THRS (arm_num, coll_type, time<, safety_gap>)`

Parameters:

arm_num :	INTEGER	[IN]
coll_type :	INTEGER	[IN]
time :	INTEGER	[IN]
safety_gap:	BOOLEAN	[IN]

Comments:

- *arm_num* is the arm on which the acquisition should be applied to.
- *coll_type* is the type of sensitivity to be acquired. This routine directly reads \$ARM_SENSITIVITY variable in relation to the specified *coll_type*. The allowed values are COLL_LOW to COLL_USER10.
- *time* is the duration in seconds of the acquisition and should correspond, at least, to the duration of the path of which the thresholds are valid (a working cycle or a single motion). The range is 1..300 seconds.

NOTE THAT, for *time* parameter, values 1 and 0 have a special meaning:

- 1 - is used to start the data acquisition to calculate the Collision Detection sensitivity thresholds
- 0 - is used to stop the data acquisition and assign the Collision Detection sensitivity thresholds.

safety_gap is an optional flag which defines if the thresholds should be calculated under a variability margin (TRUE (default value)) or exactly on an assigned path (FALSE). \$A_ALONG_2D[10, ax] predefined variable contains the variability margin value which is initialized in the configuration file.

Example:

```
ARM_COLL_THRS(1,COLL_USER7,1)
MOVE ...
...
MOVE ...
ARM_COLL_THRS(1,COLL_USER7,0)
```

See also:

Motion Programming Manual - chapter **COLLISION DETECTION** for a sample program (Program 'soglia')

11.9 ARM_COOP Built-In Procedure

The ARM_COOP built-in procedure provides the capability to switch cooperative motion on or off between two arms.

Calling Sequence: ARM_COOP (flag <, positioner_arm <, working_arm>>)

Parameters:

flag	: BOOLEAN	[IN]
positioner_arm	: INTEGER	[IN]
working_arm	: INTEGER	[IN]

Comments:

flag is set to ON or OFF in order to switch the cooperative motion on or off.
positioner_arm, if present, specifies the number of the positioner arm. If not present, \$SYNC_ARM is assumed.
working_arm, if present, represents the number of the working arm. If not specified, \$PROG_ARM is used.

Examples:

```
PROGRAM coop PROG_ARM=1
VAR p: POSITION
BEGIN
    -- program to enable cooperation between arm
    -- 1 and arm 2
    -- arm 2 is the positioner and arm 1 is the
    -- worker
    ARM_COOP(ON,2,1)
    MOVE LINEAR TO p -- arm 1 will move and arm 2
    -- will follow
    ARM_COOP(OFF,2,1)
    ARM_COOP(ON) -- error as $SYNC_ARM is not initialized
    $SYNC_ARM := 2
    -- enable cooperation between arm 2 ($SYNC_ARM)
    -- and arm 1 ($PROG_ARM)
    ARM_COOP(ON)
    ...
END coop
```

11.10 ARM_GET_NODE Built-In Function

The ARM_GET_NODE built-in function returns the node number of the next node to be processed.

Calling Sequence: ARM_GET_NODE <(arm_num)>

Return Type: INTEGER

Parameters: arm_num : INTEGER [IN]

Comments: If *arm_num* is not specified, the default program arm is used.

The value returned indicates the next path node to be processed for the arm. Path processing is ahead of the motion so this does not necessarily indicate the next path node motion.

A value of 0 will be returned if a PATH is not being processed on the arm and a value of -1 will be returned if the PATH processing has been completed. The PATH motion may not be finished when the processing is completed since PATH processing is ahead of the motion.

Examples:

```
ROUTINE skip_node(arm_num, skip_num: INTEGER)
VAR node_num : INTEGER
BEGIN
  node_num := ARM_GET_NODE(arm_num)
  WHILE(node_num <> skip_num) AND node_num > 0 DO
    node_num := ARM_GET_NODE(arm_num)
  ENDWHILE
  IF node_num > 0 THEN
    ARM_SET_NODE(skip_num + 1, arm_num)
  ELSE
    WRITE('ERROR: Can not skip the node', NL)
  ENDIF
END skip_node
```

11.11 ARM_JNTP Built-In Function

The ARM_JNTP built-in function returns the current JOINTPOS value for a specified arm.

Calling Sequence: ARM_JNTP <(arm_num)>

Return Type: JOINTPOS

Parameters: arm_num : INTEGER [IN]

Comments: If *arm_num* is not specified, the default arm is used.

Examples:

```
PROGRAM reset
VAR
  zero, robot : JOINTPOS
  i : INTEGER
BEGIN
  robot := ARM_JNTP
  FOR i := 1 TO 6 DO
    zero[i] := -robot[i]
  ENDFOR
  MOVE TO zero
END reset
```

11.12 ARM_MCOOP Built-In Procedure

Multi-cooperative means the capability for two ARMs to cooperate with the same auxiliary axes group, belonging to one of the two ARMs.

Calling Sequence: ARM_MCOOP(flag <,aux_joint <, positioner_arm <,working_arm>>)

Parameters:

flag	: BOOLEAN	[IN]
aux_joint	: INTEGER	[IN]
positioner_arm	: INTEGER	[IN]
working_arm	: INTEGER	[IN]

Comments:

flag is set to ON or OFF in order to switch the multi-cooperative motion on or off.
aux_joint is the number of the auxiliary axis. It can be omitted if *flag* is set to OFF; on the contrary, when *flag* is ON, *aux_joint* is needed.
positioner_arm, if present, specifies the number of the positioner arm. If not present, \$SYNC_ARM is assumed.
working_arm, if present, represents the number of the working arm. If not specified, \$PROG_ARM is used.

Examples:

```

PROGRAM mcoop PROG_ARM = 2
VAR xtn0001X : XTNDPOS FOR ARM[2]
VAR jnta2_1 : JOINTPOS FOR ARM[2]

BEGIN
    --
    MOVE ARM[2] TO jnta2_1

    ARM_MCOOP(ON, 7, 2, 1) --enables the multi-cooperation of
    ARms 1 and 2 on axis 7 of ARM 2
    .
    .
    MOVE LINEAR TO xtn0001X
    ARM_MCOOP(OFF, 7, 2, 1) --disables multi-cooperation
    .
    .
END mcoop

```

11.13 ARM_MOVE Built-in Procedure

This built-in procedure allows moving an axis without using a MOVE statement. It allows to activate a movement on an axis of an arm, in positive or negative direction, not synchronized with the other axes, with or without specifying a target to be reached. It also allows to deactivate an axis movement by specifying either a deceleration slope only or also a target to be reached.

Calling Sequence: ARM_MOVE(arm_num, ax_num, spd_val, acc_ovr <,final_pos>)

Parameters:

arm_num	: INTEGER [IN]
ax_num	: INTEGER [IN]
spd_val	: REAL [IN]
acc_ovr	: INTEGER [IN]
final_pos	: REAL [IN]

Comments:

- arm_num* is the arm to be moved
- ax_num* is the axis on which the built-in has to take effect
- spd_val* it indicates the maximum cruise speed for executing motion and the motion direction (positive/negative) depending on the sign. If the value is zero, it tells the built-in to deactivate motion.
- acc_ovr* it indicates the acceleration override (1..100 %) to be used calculating the acceleration/deceleration slope, during the current movement
- final_pos* it represents the target point to be reached and where the axis must stop. It can be specified both during the motion activation (in this case the movement is calculated so that such a target is reached), and in the motion deactivation.

Examples:

- 1 - If the built-in is called twice:


```
ARM_MOVE(3, 1, 0.2, 100)
this call activates motion on axis 1 of arm 3, with
maximum speed 0.2, positive direction and acceleration
slope 100%.
No target value is specified, so the movement will last
either until the stroke end (if existing) is reached, or
upon a specific request for stopping motion.
```

```
ARM_MOVE(3, 1, 0, 50, 15.23)
The current movement is deactivated; axis 1 of arm 3
must stop at the target value of 15.23 with an
acceleration slope of 50%.
```
- 2 - If the built-in is called just once:


```
ARM_MOVE(2, 7, -1.2, 80, 20)
this call activates motion on axis 7 of arm 2, with
maximum speed 1.2, negative direction and acceleration
slope 80%, and deactivates it as soon as the target
value of 20 has been reached.
The effect is that the axis keeps moving towards the
specified direction, stopping when the target is
reached. If the target cannot be reached, like in the
current example (the target is in the opposite
direction), the axis continues the movement until its
deactivation is required by means of a second ARM-MOVE
built-in call, specifying 0 as speed value, or by
indicating a different target value.
```

11.14 ARM_MOVE_ATVEL Built-in Procedure

This procedure is useful for starting the movement of an axis, for a potential infinite time, only under speed control (which means without a "MOVE" statement). The axis is seen as part of a secondary arm. The direction of the movement depends on the sign of the motor speed specified as parameter.

The system guarantees the stopping of the axis only in case of HOLD/DRIVE OFF commands; any other case has to be handled by the user via a PDL2 program.

For handling more than one axis in this way, ARM_MOVE_ATVEL should be called multiple times.

Calling Sequence: `ARM_MOVE_ATVEL(arm_num, axis_num, spd_ovr, acc_ovr)`

Parameters: arm_num : INTEGER [IN]
 axis_num : INTEGER [IN]
 spd_ovr : INTEGER [IN]
 acc_ovr : INTEGER [IN]

Comments: arm_num is the arm to be moved
 axis_num is the axis, belonging to the specified arm_num, to be moved.
 spd_ovr is the percentage (-100/0/100) of the motor speed in respect to the \$ARM_DATA[arm].MTR_SPD_LIM [axis] variable. The parameter can assume negative and positive values for inverting the motion direction.
 acc_ovr is the percentage (0-100) of the motor acceleration time in respect to variable \$ARM_DATA[arm].MTR_ACC_TIME[axis].

For stopping the axis it is needed to execute the statement:
`ARM_MOVE_ATVEL (arm_num, axis_num, 0, acc_ovr).`
 If acc_ovr is 100 the axis will be stopped in the shortest period possible; the most acc_ovr values are low, the longer is the stopping time.

Examples: `ARM_MOVE_ATVEL(2, 4, 50, 100)`
 moves axis 4 of arm 2 at 50% of the motor speed with an acceleration time equal to 100% of the characterization time.
 For stopping the axis it is needed to execute the statement:
`ARM_MOVE_ATVEL (2, 4, 0, 100).`

11.15 ARM_NUM Built-In Function

The ARM_NUM built-in function returns the arm number component of a JOINTPOS or XTNDPOS value.

Calling Sequence: `ARM_NUM (arm_value)`

Return Type: INTEGER

Parameters: arm_value : || JOINTPOS | XTNDPOS || [IN]

Comments: arm_value is the JOINTPOS or XTNDPOS for which the arm number is to be returned

Examples: `ROUTINE loader (dest:JOINTPOS)
 BEGIN
 ...
 MOVE ARM [ARM_NUM(dest)] TO dest
 ...
 END loader`

11.16 ARM_PALLETIZING Built-In Procedure

The ARM_PALLETIZING built-in procedure provides the capability to enable/disable the palletizing optional functionality, for the specified Arm.

Calling Sequence: `ARM_PALLETIZING (enb_dsb <, arm_num>)`

Parameters: enb_dsb : BOOLEAN [IN]
 arm_num : INTEGER [IN]

Comments: *enb_dsb* is set to ON in order to ENABLE the palletizing functionality; it is set to OFF in order to DISABLE it.
arm_num, if present, specifies the number of the Arm working as Palletizer. If not present, \$PROG_ARM is used.

The following errors/warnings could occur, and should be handled by the PDL2 program:

- **37168 : Position out of work zone** - this means the current position is not allowed for the Palletizing functionality
- **59446 :Robot model mismatch in requested feature** - e.g. the specified robot has a hollow wrist
- **59447 : Feature already enabled**
- **59448 : Feature not yet enabled**

Examples: PROGRAM test_arm_palletizing HOLD

```

VAR pnt0006P, pnt0007P, pnt0001P: POSITION
      pnt0002P, pnt0003P, pnt0004P, pnt0005P : POSITION

BEGIN

  $JNT_MTURN := FALSE

  ARM_PALLETIZING(ON,1)
  MOVE JOINT NEAR pnt0002P BY 300
  MOVEFLY LINEAR TO pnt0002P ADVANCE
  MOVEFLY LINEAR NEAR pnt0003P BY 20 ADVANCE
  MOVE LINEAR TO pnt0003P
  MOVEFLY LINEAR AWAY 400 ADVANCE
  MOVEFLY JOINT TO pnt0004P ADVANCE
  MOVEFLY LINEAR TO pnt0005P ADVANCE
  MOVEFLY LINEAR NEAR pnt0006P BY 300 ADVANCE
  MOVEFLY LINEAR TO pnt0006P ADVANCE
  MOVEFLY LINEAR NEAR pnt0007P BY 20 ADVANCE
  MOVE LINEAR TO pnt0007P
  MOVEFLY LINEAR AWAY 1000
  ARM_PALLETIZING(OFF,1)

END test_arm_palletizing

```

11.17 ARM_POS Built-In Function

The ARM_POS built-in function returns the current POSITION value for a specified arm.

Calling Sequence: `ARM_POS <(arm_num)>`

Return Type: POSITION

Parameters: `arm_num : INTEGER` [IN]

Comments: The returned value is relative to the current value of \$BASE, \$TOOL, \$UFRAME.

If *arm_num* is not specified, the default program arm is used.

Examples:

```

PROGRAM main
VAR
    source : POSITION EXPORTED FROM supply
    dest : POSITION
ROUTINE get_part EXPORTED FROM part_util
ROUTINE paint_part EXPORTED FROM part_util
ROUTINE release_part EXPORTED FROM part_util
BEGIN
    dest := ARM_POS(3)
    MOVE NEAR source
    get_part
    paint_part
    MOVE TO dest
    release_part
END main

```

11.18 ARM_SET_NODE Built-In Procedure

The ARM_SET_NODE built-in procedure sets the next path node to be processed.

Calling Sequence:

```
ARM_SET_NODE (node_num <, arm_num>)
```

Parameters:

node_num : INTEGER	[IN]
arm_num : INTEGER	[IN]

Comments:

If *arm_num* is not specified, the default program arm is used.
 ARM_SET_NODE sets the next arm to be processed. This is not necessarily the next PATH node motion because the processing of the PATH is ahead of the actual motion.

Examples:

```

ROUTINE skip_node (arm_num, skip_num : INTEGER)
VAR node_num : INTEGER
BEGIN
    node_num := ARM_GET_NODE(arm_num)
    WHILE (node_num <> skip_num) AND (node_num > 0) DO
        node_num := ARM_GET_NODE(arm_num)
    ENDWHILE
    IF node_num > 0 THEN
        ARM_SET_NODE(skip_num + 1, arm_num)
    ELSE
        WRITE('ERROR: Can not skip the node', NL)
    ENDIF
END skip_node

```

11.19 ARM_SOFT Built-In Procedure

The ARM_SOFT built-in procedure is used for enabling and disabling the Soft Servo modality (optional feature - for the options codes, please refer to the **C5G Control Unit Technical Specifications** Manual, chapter **Software Options**) on one or more axes (including those that are subject to gravity) of a certain arm.

This feature is used in some applications when it is required that the robot is compliant to movements produced by external forces. For example, when a workpiece is hooked from a press, the detachment is facilitated by the pushing of a roll; the robot must follow the movement without opposition.

When the Soft Servo modality is enabled, the robot should be steady. The Soft Servo

modality is automatically disabled by a DRIVE OFF. The Soft Servo modality works fine only when the dynamic model algorithm (Fast Move) is active.

It is strongly recommended to disable the Collision Detection feature before calling ARM_SOFT (ON, ...). Collision Detection can be enabled again after calling ARM_SOFT (OFF).

The degree of compliance of each axis must be defined when this feature is enabled. A value of 100 indicates that the brakes for that axis should be completely released; 50 means half-released; 0 means totally blocked.

Calling Sequence: `ARM_SOFT (flag <, ax1, ax2, ax3, ax4, ax5, ax6, arm_num>)`

Parameters:

<code>flag</code> : BOOLEAN	[IN]
<code>ax1, ax2, ax3, ax4, ax5, ax6</code> : INTEGER	[IN]
<code>arm_num</code> : INTEGER	[IN]

Comments: *Flag* is used for enabling (ON) and disabling (OFF) the Soft Servo modality.
ax1, ax2, ax3, ax4, ax5, ax6 are the compliance degree of each axis. These parameters should only be specified when enabling the Soft Servo modality.
arm_num is an optional parameter containing the arm number. If not specified, the default program arm is used.

Examples:

```

MOVE LINEAR TO pnt0001p
ARM_SOFT (ON, 100, 100, 20, 1, 0, 1)
  -- in this section the arm is enabled to move under the
  -- effect of external strengths
ARM_SOFT (OFF)

```



\$TOOL_MASS: Mass of the tool and **\$TOOL_CNTR:** Tool center of mass of the tool of the related arm must be properly initialized before enabling the Soft Servo modality otherwise the correctness of robot movements are not guaranteed.

11.20 ARM_XTND Built-In Function

The ARM_XTND built-in function returns the current XTNDPOS value for a specified arm.

Calling Sequence: `ARM_XTND <(arm_num)>`

Return Type: XTNDPOS

Parameters: `arm_num` : INTEGER [IN]

Comments: If *arm_num* is not specified, the default arm is used.

Examples:

```

CURXPOS := ARM_XTND
MOVE TO checkxpos
MOVE TO curxpos

```

11.21 ARRAY_DIM1 Built-In Function

The ARRAY_DIM1 built-in function returns the size of the first dimension of an ARRAY.

Calling Sequence: `ARRAY_DIM1 (array_val)`

Return Type: INTEGER

Parameters: array_val : ARRAY [IN]

Comments: array_val can be an ARRAY of any type and dimension.

Examples:

```

ROUTINE print_ary(partlist : ARRAY OF INTEGER)
VAR i, size : INTEGER
BEGIN
    size := ARRAY_DIM1(partlist)
    FOR i := 1 TO size DO
        WRITE('Element ', i, ' --> ', partlist[i], NL)
    ENDFOR
END print_ary
    
```

11.22 ARRAY_DIM2 Built-In Function

The ARRAY_DIM2 built-in function returns the size of the second dimension of an ARRAY.

Calling Sequence: ARRAY_DIM2 (array_val)

Return Type: INTEGER

Parameters: array_val : ARRAY [IN]

Comments: array_val must be a two dimensional array. If a one dimensional array is used an error occurs.

Examples:

```

ROUTINE print_2dim_ary(matrix : ARRAY[* , *] OF INTEGER)
VAR i, j, size1, size2 : INTEGER
BEGIN
    size1 := ARRAY_DIM1(matrix)
    size2 := ARRAY_DIM2(matrix)
    FOR i := 1 TO size1 DO
        FOR j := 1 TO size2 DO
            WRITE('Element ', i, ' ', j, ' --> ', matrix[i, j], NL)
        ENDFOR
    ENDFOR
END print_2dim_ary
    
```

11.23 ASIN Built-In Function

The ASIN built-in function returns the arc sine of the argument.

Calling Sequence: ASIN (number)

Return Type: REAL

Parameters: number : REAL [IN]

Comments: The arc sine is measured in degrees. The result is in the range of -90 to 90 degr. number specifies a real number in the range of -1 to 1.

Examples:

```

ASIN(0.5)      -- result is 30
ASIN(-0.5)     -- result is -30
    
```

11.24 ATAN2 Built-In Function

The ATAN2 built-in function calculates the arc tangent of a quotient.

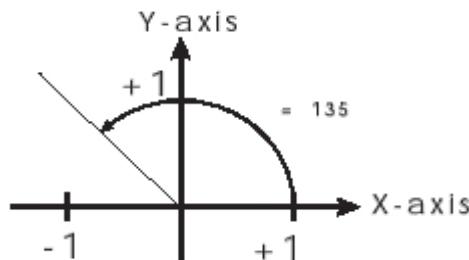
Calling Sequence: ATAN2 (y, x)

Return Type: REAL

Parameters: y : REAL [IN]
x : REAL [IN]

Comments: The arc tangent is measured in degrees.
The result is in the range of -180 to 180.
The quadrant of the point (x, y) determines the sign of the result.
If x and y are both zero an error occurs.

Examples: x := ATAN2(0, 0) -- ERROR
x := ATAN2(1, -1) -- x = 135 (see diagram)



11.25 AUX_COOP Built-In Procedure

The AUX_COOP built-in procedure enables or disables the cooperative motion between a robot and a positioner. The positioner is defined as a group of auxiliary axes, not a second arm.

Calling Sequence: AUX_COOP (flag <,aux_joint> <,arm_num>)

Parameters: flag : BOOLEAN [IN]
aux_joint : INTEGER [IN]
arm_num : INTEGER [IN]

Comments: *flag* is set to ON or OFF in order to switch the auxiliary cooperative motion on or off.
aux_joint is the number of the auxiliary axis. It can be omitted if *flag* is set to OFF; on the contrary, when *flag* is ON, *aux_joint* is needed.
arm_num, if present, represents the number of the arm on which the auxiliary cooperative motion should be executed. If not specified, \$PROG_ARM is used.

Examples:

```

PROGRAM aux
VAR x: XTNDPOS
-- Assume that this program is in a system with a 5 axes
-- robot, a table positioner with 2 axes (axis 6 and 7)
-- and an additional 1-axis positioner (axis 8).
BEGIN
    -- enable the cooperative motion between the robot
    -- and the first positioner
    AUX_COOP(ON,7)
    MOVE LINEAR TO x -- the robot will move cooperatively
  
```

```

-- enable the cooperative motion between the robot
-- and the second positioner
AUX_COOP(ON,8,1)
-- disable the cooperative motion between the robot and
-- all the positioners
AUX_COOP(OFF)
END aux

```

11.26 AUX_DRIVES Built-In Procedure

The AUX_DRIVES built-in procedure is used when the user needs to switch the positioner between DRIVE OFF and DRIVE ON. The positioner is defined as a group of auxiliary axes. When such a built-in procedure is activated in DRIVE OFF state, a hardware safety device is present (C5G-IESK).

Note that the program which switches to DRIVE ON must be **the same** which switches to DRIVE OFF.



- To execute AUX_DRIVES procedure, the ARM
 - must be NOT MOVING and
 - must have NOT ANY PENDING MOVEMENTS.
- Note that this procedure could take A COUPLE OF SECONDS to be accomplished.

Calling Sequence: AUX_DRIVES (enable_flag, axis_num <,arm_num>)

Parameters:

enable_flag	: BOOLEAN	[IN]
axis_num	: INTEGER	[IN]
arm_num	: INTEGER	[IN]

Comments: enable_flag enables and disables the selective DRIVE ON on the specified axis. If ON, the DRIVE ON is enabled
axis_num is one of the axes defining the arm



If '0' it means ALL robot axes which are NOT specified as AUXILIARY axes.

arm_num is the arm including the being enabled/disabled axis.

Examples:

```

AUX_DRIVES (ON, 7, 1) -- enables the first arm, linked to axes 7 and 8
AUX_DRIVES (OFF, 9, 2) -- disables the second arm, linked to axes 9 and 10
MOVE TO pnt0001j -- the robot (axes 1..6) and the first arm
(axes 7..8) are moved

```

11.27 AUX_SET Built-In Procedure

The AUX_SET built-in procedure is used when the robot must change the electrical welding gun. The motor current and the resolver reading of the electrical welding gun must be disabled. After the new electrical welding gun has been recognized by the program, the motor currents and the resolver reading must be reenabled.

Calling Sequence: AUX_SET (flag <,aux_axis <,arm_num>>)

Parameters:

flag	: BOOLEAN	[IN]
aux_axis	: INTEGER	[IN]
arm_num	: INTEGER	[IN]

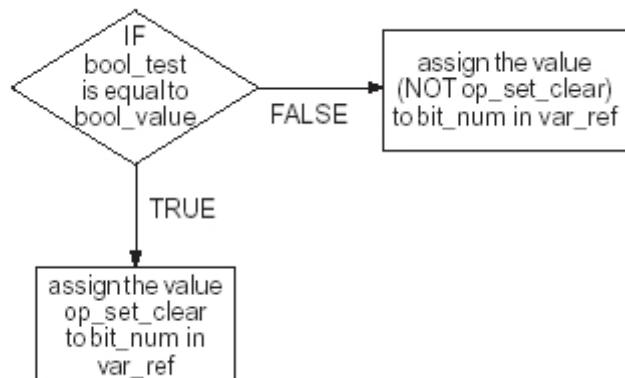
- Comments:** *flag* can assume the value ON or OFF.
aux_axis is used for indicating the axis of which the motor and the resolver must be disconnected (case of flag set to OFF) or connected (case of flag set to ON). If not specified, the first axis declared as electrical welding gun is referred.
arm_num indicates the arm number to which the electrical welding gun belongs. If not specified, the \$PROG_ARM is used.
- Examples:**
- ```
AUX_SET(OFF, 8, 1)
AUX_SET(ON, 8)
```

## 11.28 BIT\_ASSIGN Built-In Procedure

The BIT\_ASSIGN built-in procedure assigns the value of 1 or 0 to a bit of an INTEGER variable or port. The value to be assigned to the bit is the result of a comparison between BOOLEAN parameters passed to this built-in.

- Calling Sequence:**
- ```
BIT_ASSIGN (var_def, bit_num, bool_test
<, op_set_clear <, bool_value >>)
```
- Parameters:**
- | | | |
|----------------------|---------|-----------|
| <i>var_ref:</i> | INTEGER | [IN, OUT] |
| <i>bit_num:</i> | INTEGER | [IN] |
| <i>bool_test:</i> | BOOLEAN | [IN] |
| <i>op_set_clear:</i> | BOOLEAN | [IN] |
| <i>bool_value:</i> | BOOLEAN | [IN] |
- Comments:**
- var_ref* is an INTEGER variable or port reference. A bit of this variable will be set by this built-in.
- bit_num* is an INTEGER expression indicating the bit to be set. The value must be in the range from 1 to 32, where 1 corresponds to the least significant bit of the INTEGER.
- The BIT_ASSIGN built-in procedure sets the bit specified by *bit_num* in the variable *var_ref* to a value that is the result of the comparison between the *bool_test* BOOLEAN variable (or port) and the *bool_value* BOOLEAN value.
- If *bool_test* is equal to *bool_value*, *bit_num* in *var_ref* will assume the value specified in *op_set_clear*; on the contrary, *bit_num* will be set to the negated value of *op_set_clear*.
- op_set_clear* and *bool_value* are optional parameters. If not specified, their default value is TRUE.

The following figure shows the execution of the BIT_ASSIGN built-in procedure.



The BIT_ASSIGN built-in procedure can be used as an action in a condition handler. In this case, the values of *op_set_clear* and *bool_test* are evaluated in the moment in which the condition is defined and not when the action is executed.

Examples:

```
-- if $DOUT[20] is TRUE, bit 3 of value is set to FALSE
-- else to TRUE
BIT_ASSIGN(value, 3, $DOUT[20], FALSE, TRUE)

-- if bool_var is TRUE, bit 2 in $AOUT[4] is set to TRUE
-- else to FALSE
BIT_ASSIGN($AOUT[4], 2, bool_var)

BIT_ASSIGN(int_var, 7, bool_var, bool2_var, bool3_var)
```

11.29 BIT_CLEAR Built-In Procedure

The BIT_CLEAR Built-In procedure clears a bit of an INTEGER variable.

Calling Sequence: `BIT_CLEAR(var_ref, bit_num)`

Parameters: `var_ref : INTEGER` [IN, OUT]
`bit_num : INTEGER` [IN]

Comments: `var_ref` is an INTEGER variable or port reference.
`bit_num` is an INTEGER expression indicating the bit to be cleared. The value must be in the range 1 to 32 where 1 corresponds to the least significant bit of the INTEGER.
The BIT_CLEAR Built-In procedure clears the bit specified by `bit_num` in the variable `var_ref`.

The BIT_CLEAR built-in procedure can be used as an action in a condition handler. The values of the parameters are evaluated at the condition definition time and not at action execution time. For more information, refer to [Condition Handlers](#) chapter.

Examples:

```
BIT_CLEAR(value, 1)
BIT_CLEAR(value, bit_num)
BIT_CLEAR($WORD[2], bit_num)
```

11.30 BIT_FLIP Built-In Function

The BIT_FLIP Built-In function can be used for detecting the positive or negative transition of a bit in one of the following analogue ports: \$AIN, \$AOUT, \$WORD, \$FMI, \$FMO.

This function can be used as condition expression in a CONDITION handler or in a WAIT FOR statement. The BIT_FLIP cannot be used as a normal statement in the program body.

Calling Sequence: `BIT_FLIP(port, bit_num, bit_state)`

Return Type: BOOLEAN

Parameters: `port: INTEGER` [IN]
`bit_num: INTEGER` [IN]
`bit_state: BOOLEAN` [IN]

Comments: *port* is the INTEGER analogue port reference whose bit is to be tested.
bit_num is an INTEGER value specifying the bit to be tested. The value must be in the range 1 to 32 where 1 corresponds to the least significant bit of the INTEGER. To be noted that the bit value is evaluated at CONDITION definition time and not during the scanning of the expression.
bit_state is a BOOLEAN value indicating if a positive (TRUE, ON) or negative (FALSE, OFF) transition should be tested.

Examples:

```

CONDITION[1] :
  WHEN BIT_FLIP ($WORD[100], 4, ON) DO
    HOLD
  ENDCONDITION
  ENABLE CONDITION[1]
  .....
  WAIT FOR ($USER_BYTE[3], 5, FALSE)

```

11.31 BIT_SET Built-In Procedure

The BIT_SET Built-In procedure sets a bit of an INTEGER variable.

Calling Sequence: BIT_SET(*var_ref*, *bit_num*)

Parameters: *var_ref* : INTEGER [IN, OUT]
bit_num : INTEGER [IN]

Comments: *var_ref* is an INTEGER variable or port reference.
bit_num is an INTEGER expression indicating the bit to be set. The value must be in the range 1 to 32 where 1 corresponds to the least significant bit of the INTEGER.
The BIT_SET Built-In procedure sets the bit specified by *bit_num* in the variable *var_ref*.
The BIT_SET built-in procedure can be used as an action in a condition handler. The values of the parameters are evaluated at the condition definition time and not at action execution time. For more information, refer to [Condition Handlers](#) chapter.

Examples:

```

BIT_SET(value, 1)
BIT_SET(value, bit_num)
BIT_SET($WORD[2], bit_num)

```

11.32 BIT_TEST Built-In Function

The BIT_TEST Built-In function returns a BOOLEAN value indicating whether a bit of an INTEGER is set or cleared.

Calling Sequence: BIT_TEST(*test_val*, *bit_num* <, *bit_state*>)

Return Type: BOOLEAN

Parameters: *test_val* : INTEGER [IN]
bit_num : INTEGER [IN]
bit_state : BOOLEAN [IN]

Comments: *test_val* is the INTEGER value whose bit is to be tested. *test_val* can be an expression, a user defined variable reference, or a system port reference.
bit_num is an INTEGER value specifying the bit to be tested. The value must be in the range 1 to 32 where 1 corresponds to the least significant bit of the INTEGER.

bit_state is a BOOLEAN value indicating the desired bit state to test for. If not specified, ON is assumed.

If the BIT_TEST Built-In function is used in a condition handler, the *bit_state* parameter must be specified. This is also true when using BIT_TEST in a WAIT FOR statement.

The BIT_TEST Built-In function returns TRUE if the bit in *test_val* which is specified by *bit_num* is currently set to the value specified by *bit_state*.

Examples:

```
bool_var := BIT_TEST($WORD[index], bit_num)
bool_var := BIT_TEST(test_val, bit_num)
bool_var := BIT_TEST(test_val, bit_num, FALSE)
```

11.33 CHR Built-In Procedure

The CHR built-in procedure assigns a character, specified by its numeric code, to a STRING at an indexed position.

Calling Sequence: CHR (str, index, char_code)

Parameters:	str : STRING	[IN, OUT]
	index : INTEGER	[IN]
	char_code : INTEGER	[IN]

Comments:

str is the STRING variable to receive the character.

index is the position in *str* where the character is assigned.

char_code is the numeric code used to specify the character.

If *str* is uninitialized and *index* is one, then *str* is initialized to a length of one having a value equal to the character.

If it is initialized, any position in *str* can be indexed.

str can also be extended by one character if the new length is still in the range of the declared physical length of the STRING. If extending *str* would exceed the declared length, then *str* is not modified.

Examples:

```
PROGRAM chr_test
VAR
dest_string : STRING[5]
indx, code : INTEGER
BEGIN
dest_string := 'ACCD'
code := 66 -- ASCII code for 'B'
indx := 2
CHR(dest_string, indx, code) -- dest_string equals 'ABCD'
code := 69 -- ASCII code for 'E'
indx := 5
CHR(dest_string, indx, code) -- dest_string equals 'ABCDE'
code := 70 -- ASCII code for 'F'
indx := 6
CHR(dest_string, indx, code) -- dest_string still equals
-- 'ABCDE' (no error)
END chr_test
```

11.34 CLOCK Built-In Function

The CLOCK built-in function returns the current time.

Calling Sequence: CLOCK

Return Type:	INTEGER
Comments:	The current time is returned in seconds counted from January 1, 1980. For example, a value of 0 indicates midnight on December 31, 1979. CLOCK is typically used to measure differences in time. If the value is negative, the hardware clock has never been set. The value is actually incremented every other second.
Examples:	<pre>cur_time := CLOCK WRITE(CLOCK, NL) -- say time is 1514682000 -- 30 second time interval WRITE(CLOCK, NL) -- now time is 1514682030</pre>

11.35 COND_ENABLED Built-In Function

The COND_ENABLED built-in function tests a condition to see if it is enabled and returns a BOOLEAN result.

Calling Sequence:	COND_ENABLED (cond_num <, prog_name >)
Return Type:	BOOLEAN
Parameters:	cond_num : INTEGER [IN] prog_name: STRING [IN]
Comments:	<p><i>cond_num</i> is the number of the condition to be tested.</p> <p><i>prog_name</i> is the name of the program to which the condition <i>cond_num</i> belongs. If not specified, the program calling this built-in is used.</p> <p>The returned value will be TRUE if the condition <i>cond_num</i> is enabled (locally or globally) and FALSE if the condition is disabled. An error occurs if the condition is not defined</p>
Examples:	<pre>-- condition 3 of program pippo is tested bool_var := COND_ENABLED(3, pippo) -- condition 5 of the executing program is tested bool_var := COND_ENABLED(5) -- check if condition 1 is enabled for disabling it IF COND_ENABLED(1) THEN DISABLE CONDITION[1] ENDIF</pre>

11.36 COND_ENBL_ALL Built-In Procedure

The COND_ENBL_ALL built-in procedure returns the current state (enabled or disabled) of the conditions that have been defined by a certain program.

Calling Sequence:	COND_ENBL_ALL (cond_map <, prog_name >)
Parameters:	cond_map : ARRAY OF INTEGER [IN, OUT] prog_name: STRING [IN]
Comments:	<p><i>cond_map</i> is an array of at least 9 elements that, after this built-in procedure has been executed, will contain the bit mapping of the enabled condition handlers associated to the program. For each array element, only bits from 1 to 30 are used. If the bit is set to 1, it means that the corresponding condition is enabled. If the bit has the value of 0, this means that the condition is not defined or not enabled. A similar kind of mapping is used in \$PROG_CONDS program</p>

stack variable, but this contains the information of which condition handlers are defined by the program. Using this built-in it is possible to check which condition handlers are enabled.

prog_name is the name of the program that owns the conditions. If not specified, the program calling this built-in is used.

This built-in allows to get from PDL2 the same information that the user can have by issuing the system command PROGRAM VIEW /FULL, where the enable conditions are marked with a wildcard.

Examples:

```

PROGRAM enball NOHOLD
VAR al_enbl_chh : ARRAY [15] OF INTEGER
    i, j, mask : INTEGER
    s : SEMAPHORE NOSAVE
BEGIN
    -- get information about enabled conditions of enball
    program
        COND_ENBL_ALL(al_enbl_chh, 'enball')
        WRITE (NL)
        -- only the first 9 elements of $PROG_CONDS and
        -- al_enbl_chh ar
        FOR i := 1 TO 9 DO
            mask := j := 1
            WHILE mask AND 0xFFFFFFFF <> 0 DO
                -- see if cond is defined
                IF $PROG_CONDS[i] AND mask <> 0 THEN
                    WRITE (' Condition ', (i - 1) * 30 + j::3, 'is defined')
                -- see if it is enabled
                IF al_enbl_chh[i] AND mask <> 0 THEN
                    WRITE (' and enabled', NL)
                ELSE
                    WRITE (NL)
                ENDIF
                ENDIF
                mask := mask SHL 1
                j := j + 1
            ENDWHILE
        ENDFOR
        WAIT s
    END enball

```

See also:

[Predefined Variables List](#) chapter (\$PROG_CONDS) and [Condition Handlers](#) chapter

11.37 CONV_TRACK Built-In Procedure

The CONV_TRACK built-in procedure initializes, activates, deactivates the conveyor tracking optional application.

Calling Sequence: CONV_TRACK (cmd, num_tbl <, arm_num <, ofst>>)

Parameters:	cmd: INTEGER	[IN]
	num_tbl: INTEGER	[IN]
	arm_num: INTEGER	[IN]
	ofst: REAL	[IN]

Comments: cmd command for conveyor tracking; the following values are available:

- -1 initialize (CONV_INIT)
- 0 deactivate (CONV_OFF)

- 1 initialize and activate (CONV_INIT_ON)
 - 2 activate (CONV_ON)
- num_tbl* table number configured for conveyor
arm_num is the number of the Arm which is involved in the conveyor tracking functionality. If not specified, *prog_arm* is used. Refer to the **Motion Control** chapter in the **Motion Programming** manual for further details
ofst is the distance between the sensor and the being tracked object, at the conveyor initialization time; the default value is zero.

11.38 COS Built-In Function

The COS built-in function returns the cosine of a specified angle.

Calling Sequence: COS (angle)

Return Type: REAL

Parameters: angle : REAL [IN]

Comments: angle is specified in degrees.

The returned value will be in the range of -1.0 to 1.0.

Examples: x := COS(87.4) -- x = 0.04536

x := SIN(angle1) * COS(angle2)

11.39 DATE Built-In Function

The DATE built-in function returns the current date or a date corresponding to a specified time.

Calling Sequence: DATE <(date_in)>

Return Type: STRING

Parameters: date_in : INTEGER [IN]

Comments: The date is returned in the format day-month-year. Where day is 2 characters, month is 3 characters and year is 2 characters. Each group of characters are separated by a hyphen ('-'). The minimum string size is 9 characters, needed for storing the day, the month and the year. For getting also hour, or minutes and seconds, the string must be declared of 20 characters. If the return value is assigned to a variable whose maximum length is less than 9, the result will be truncated.

date_in must be passed in integer format according to the table shown below. See the example to better understand how to set up an input date value

7 bits	4 bits	5 bits	5 bits	6 bits	5 bits
YEAR	MONTH	DAY	HOUR	MINS	SECS

The following conditions apply to the fields above:

YEAR:	The passed value represents the desired year minus 1980. (To pass 1994 for example, the year field would be 1994-1980, or 14. To pass 1980, the year field should be zero.)
MONTH:	A value from 1 to 12
DAY:	A value from 1 to 31
HOUR:	A value from 0 to 23
MINS:	A value from 0 to 59
SECS:	A value from 0 to 29, the effective numbers of seconds divided by 2

If *date_in* is not specified, the current date is returned. Otherwise, the date corresponding to *date_in* is returned

Examples:

```
PROGRAM pr_date
VAR date_str : STRING[20]
    old_time : INTEGER
BEGIN
    old_time := 0b0001110011111000101110011001010
    .
    .
    date_str := DATE
    WRITE('Current DATE (dd-mmm-yy) = ', date_str, NL)
    WRITE('Old DATE = ', DATE(old_time), NL)
-- should be 28-JUL-94
END pr_date
```

11.40 DIR_GET Built-In Function

This routine is used for getting the directory of the specified program.

Calling Sequence:

```
directory := DIR_GET<(mode,prog_name)>
```

Return Type:

STRING

Parameters:

mode: INTEGER	[IN]
prog_name: STRING	[IN]

Comments:

mode is an optional parameter that indicates the modality of usage for this built-in function. Possible values are: 0, the active executing directory is returned; 1, the directory from which the program code has been loaded is returned; 2, the directory from which the variables have been loaded is returned. *prog_name* is the name of the program the directory of which is to be searched. If not specified, the program that executes the routine is used.

Examples:

```
ac_dir := DIR_GET -- gets the current executing directory --
of the calling program
ac_dir_cod := DIR_GET(1, 'pippo') -- gets the directory from
-- where the code of pippo has been loaded
```

11.41 DIR_SET Built-In Procedure

This built-in procedure allows changing the working directory of the executing program.

Calling Sequence:

```
DIR_SET (new_dev_dir)
```

Parameters:

new_dev_dir: STRING	[IN]
---------------------	------

Comments: *new_dev_dir* indicates the directory to be assigned as the working directory.
PLEASE NOTE THAT, for positioning at the root of a device, it is needed to insert the \ character after the device name, otherwise the device will be accessed at the current working directory.

For example, if the current working directory is UD:\app1\arc, the following statements:

```
DIR_SET('UD:')
SYS_CALL('FUDM', 'prova')
will create UD:\app1\arc\prova.
```

For creating a directory at the root of UD:, the user will have to write

```
DIR_SET('UD:\\')
In the example, a statement
SYS_CALL('FUDM', 'prova')
will create the directory UD:\prova.
```

11.42 DRIVEON_DSBL Built-In Procedure

This routine can be used for disabling some axes to the DRIVE ON command.

Calling Sequence: DRIVEON_DSBL (arm_num<, disable_axis1, ...disable_axis10>)

Parameters: arm_num : INTEGER [IN]
 disable_axis1...disable_axis10: BOOLEAN [IN]

Comments: *arm_num* is the arm of interest.
disable_axis1..10 are optional BOOLEAN parameters. If passed as TRUE, the axis is disabled to the DRIVE ON effect. If one or more parameters are not specified, it is treated as FALSE.

This function must be called when the state of the DRIVEs is OFF.

Examples: DRIVEON_DSBL (1, FALSE, TRUE) -- axis 2 is disabled upon DRIVE ON

11.43 DV_CTRL Built-In Procedure

The DV_CTRL built-in procedure is used to control a communication device.

Calling Sequence: DV_CTRL (code <, param>...)

Parameters: code : INTEGER [IN]
 param : data type specified by code [IN/OUT]

Comments: *code* is an integer expression that specifies the desired command to be performed.
param is the list of parameters required for the command specified by *code*.

The following outlines the valid values for code and their expected parameters:

- 1 Create a PIPE
 Parameters:
 - a STRING for the pipe name
 - an INTEGER for the buffer size (default = 512)
 - an INTEGER for the number of senders (default = 1)
 - an INTEGER for the number of readers (default = 1)
 - an INTEGER for flags (0x1 means deleting the pipe when no longer opened)

- 2 Delete a PIPE
Parameters:
 - a STRING for the pipe name
- 3 Get the port characteristics
Parameters:
 - a STRING for the port name
 - an INTEGER for the port characteristics
 - an INTEGER for the size of the read-ahead buffer
- 4 Set the port characteristics
Parameters:
 - a STRING for the port name
 - an INTEGER for the port characteristics
 - an INTEGER for the size of the read-ahead buffer
- 5 Add a router to the network
Parameters:
 - a STRING for the destination. Can be a name or an IP address (dotted notation like 177.22.100.201). If this parameter is 0 it defines the default Gateway
 - a STRING for the Gateway
- 6 Delete a router in the network
Parameters:
 - a STRING for the destination. Can be a name or an IP address (dotted notation like 177.22.100.201). If this parameter is 0 it defines the default Gateway
 - a STRING for the Gateway
- 7 Add a host to the network
Parameters:
 - a STRING for the host name
 - a STRING for the host address
- 8 Delete a host in the network
Parameters:
 - a STRING for the host name
 - a STRING for the host address
- 9 Get the IP address of the host from the name
Parameters:
 - a STRING for the host name
 - a STRING (by reference) for storing the host address
- 10 Get the host name given the IP address
Parameters:
 - a STRING for the host address
 - a STRING (by reference) for storing the host address
- 11 Get the current IP address and the subnet mask
Parameters:
 - a STRING (by reference) for the IP address
 - a STRING (by reference) for the subnet mask
 - a STRING (by reference) for the host name
- 12 Set the current IP address and the subnet mask
Parameters:
 - a STRING for the IP address
 - a STRING for the subnet mask
 - a STRING for the host name
- 13 reserved

- 14 This routine returns the index of the I/O entry in the \$IO_STS table.
 Parameters:
 - a STRING which is the name of the I/O Point
 - an INTEGER which is the index of the I/O entry in the \$IO_STS table
- 15 This routine returns the index of the Device entry in \$IO_DEV table
 Parameters:
 - a STRING which is the name of the Device
 - an INTEGER which is the index of the Device entry in \$IO_DEV table
- 16 Connect/Disconnect a device on a fieldbus network
 Parameters:
 - a STRING which is the name of the device
 - an INTEGER which is the connect flag:
 ConnectFlag = 1 --> Connect , ConnectFlag = 0 --> Disconnect
 - an optional INTEGER which is the Timeout: if set to -1 asynch, if set to zero default, if > 0 timeout value
- 20 Configure e-mail functionality.
 Parameters:
 - An INTEGER which is the given back configuration ID
 - An ARRAY[6] of STRINGs[63] containing the configuration strings:
 • [1] incoming mail server
 • [2] outgoing mail server
 • [3] receiver ID
 • [4] login
 • [5] password
 • [6] name of the subdirectory of UD:\sys\email containing the attached files
 - An ARRAY[5] of INTEGERS containing the configuration integers:
 • [1] flags: 1 - enable, 32 - do not delete the attachment directory at startup time
 • [2] maximum allowed e-mail size
 • [3] polling interval for POP3 server (not used)
- 21 Send e-mail.
 Parameters:
 - An INTEGER which is the configuration ID
 - An ARRAY[2,8] of STRINGs[63] which includes "To:" and "CC:" fields contents:
 • [1] array of "To:" contents
 • [2] array of "CC:" contents
 - A STRING[1023] containing the e-mail title
 - A STRING[1023] containing the e-mail body
 - An ARRAY[10] of STRINGs[63] containing the attached files list
 - An INTEGER containing the e-mail priority
- 22 Read the number of e-mails currently waiting on POP3 server.
 Parameters:
 - An INTEGER containing the configuration ID
 - An INTEGER containing the given back number of e-mails on the server
- 23 Receive e-mails.**Note that when an e-mail is read, it is not deleted on POP3 server.**
 Parameters:

- An INTEGER containing the configuration ID
- An INTEGER containing the index of the e-mail to be received (1=the least recent)
- A STRING[63] containing the "From:" field address
- A STRING[1023] containing the e-mail title
- A STRING[1023] containing the e-mail body
- An INTEGER containing the receipt date (ANSI TIME)
- An ARRAY[2,8] of STRINGs[63] which include "To:" and "CC:" fields contents:
 - [1] array of "To:" contents
 - [2] array of "CC:" contents
- An ARRAY[10] of STRINGs[63] containing the attached files list

24 Delete e-mails from the server.



The message index must be between 1 and the number of e-mails currently on the POP3 server (such a value is given back by DV_CNTRL 22 built-in routine). Such indices are consistent until the messages are not deleted from the server

Parameters:

- An INTEGER containing the configuration ID
- An INTEGER containing the index of the message to be deleted

25 Get an e-mail header. This command reads the e-mail header without downloading the message from the server.

Parameters:

- An INTEGER containing the configuration ID
- An INTEGER containing the index of the e-mail whose header is to be read
- A STRING[63] containing the "From:" field address
- A STRING[1023] containing the e-mail title
- An INTEGER containing the server receipt date (ANSI TIME)
- An ARRAY[2,8] of STRINGs[63] which include "To:" and "CC:" fields contents:
 - [1] array of "To:" contents
 - [2] array of "CC:" contents

26 Close the e-mail channel (e-mail ID).

Parameters:

- An INTEGER containing the configuration ID

27 Get information about a network connection.

Parameters:

- An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
- A STRING (by reference) for the session peer
- A STRING (by reference) for the accept peer
- A STRING (by reference) for the connect address
- An INTEGER (by reference) for the remote port
- An INTEGER (by reference) for the local port
- An INTEGER (by reference) for options
- An INTEGER (by reference) for the linger time

- 28 This code can be used for configuring different aspects of a TCP/IP channel. The parameters to the routine are used for specifying which feature is to be enabled or cleared or in fact not touched at all.
 Parameters:
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
 - An INTEGER which is the value (either 0 or 1) to be assigned to the current bit. See the [Examples](#) below.
 - An INTEGER which indicates the features involved in the required modification (-1 means restore to default values); see the [Examples](#) below. The following bits are available:
 - 1 - NO_DELAY
 - 2 - NOT_KEEPALIVE - do NOT send challenge packets to peer, if idle
 - 4 - LINGER
 - 8 - UDP_BROADCAST - permit sending of broadcast messages
 - 16 - OOB - send out-of-band data
 - 32 - DONT_ROUT - send without using routing tables
 - 64 - NOT_REUSEADDR - do not reuse address
 - 128 - SIZE_SENDBUF - specify the maximum size of the socket-level "send buffer"
 - 256 - SIZE_RCVBUF - specify the maximum size of the socket-level "receive buffer"
 - 512 - OOB_INLINE - place urgent data within the normal receive data stream.
-  The listed above features are the standard ones for TCP/IP networks; for further information, please refer to the descriptions provided on the Internet.
- 29 Accept on a port for a connection.
 Parameters:
 - An INTEGER for the linger time
 - An INTEGER for the size of the "send buffer"
 - An INTEGER for the size of the "receive buffer"
- 30 Establish a TCP/IP connection.
 Parameters:
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
 - An INTEGER for the local port number
 - A STRING Subnet mask of clients that will accept (optional 0.0.0.0 means any)
- 31 Disconnect a TCP/IP connection.
 Parameters:
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
- 32 Obtain statistics about the network LAN.
 Parameters:

- An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
 - An ARRAY[8] of INTEGERs containing the statistics
- 33 Clear the statistics about the network LAN.
 Parameters:
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
- 41 Copy from string to variable - serialize/deserialize.
 Parameters:
 - A BOOLEAN indicating True=Serialize (Record->String), False=Deserialize(String->Record)
 - A STRING which is the Source of the data when deserializing or the Destination when serializing
 - A RECORD which is the variable to be filled in
 - A STRING indicating the name of the field
 - An INTEGER indicating the Offset in the string (1+)
 - An INTEGER indicating the Size of the being copied data
 - An INTEGER which indicates the features involved in the required operation:
 - 1 - unsigned char, shorts
 - 2 - big endian
 - 4 - zero memory first
- 42 Obtains a list of devices suitable for specific operations
 Parameters:
 - An INTEGER indicating the Operation Flag:
 - 1 = Mount
 - 2 = Dismount
 - 3 = Device Set
 - 4 = Device Get
 - 5 = Filer Copy Source
 - 6 = Filer Copy Destination
 - 7 = Filer Delete
 - 8 = Filer Print Source
 - 9 = Filer Print Destination
 - 10 = Filer Rename
 - 11 = Filer Translate
 - 12 = Filer View
 - 13 = Filer Utility Attribute
 - 14 = Filer Utility Attribute Set
 - 15 = Modem
 - 16 = PPP
 - 17 = Remote Mount
 - 18 = Backup Destination / Restore Source
 - 19 = Backup Source /Restore Destination
 - An ARRAY OF STRINGs indicating the returned list of devices
 - An optional STRING for the Username

Examples

Use of code 28:

if the user would like both to enable the NO_DELAY functionality and disable the NOT_KEEPALIVE functionality, the DV_CNTRL built-in routine is to be called as follows:

```
DV_CNTRL (28, dev_lun, 0x1, 0x3)
```

where:

- 0x3 means that the features which are involved in the modification are: NO_DELAY (1) and NOT_KEEPALIVE (2); note that the other bits are ignored.
- 0x1 means bit 1 is to be set and bit 2 is to be cleared.

11.44 DV_STATE Built-In Function

This function returns information about a specified device.

Calling Sequence: DV_STATE (dev_name_str)

Return Type: INTEGER

Parameters: dev_name_str : STRING [IN]

Comments: dev_name_str is the device name for which information is to be retrieved. The dev_name_str must be the logical name used for the protocol which is mounted on the device. Each bit or group of bits, in the returned value, indicates some information about the device. A returned value of 0, indicates that the device does not exist. Here follows the meaning of the returned INTEGER:

Bits	Value	Meaning
1	1	the device is attached
2	1	a file is opened on the device
3-5		for the protocol mounted on the device
	0:	Default protocol
	1:	reserved
	2:	Winc5g on PC protocol (serial or network)
	3:	3964r
6-8	--	reserved
9-15		for the type of device
	0	for NULL device
	1	for a Window device
	2	a file device
	3	a serial line device with no protocol mounted on it
	4	reserved
	5	a serial line under a protocol
	6	reserved
	7	a pipe device
	8	Winc5g on serial connection
	9	3964r protocol
	10	FTP Network protocol

11

Winc5g on Ethernet connection

16-32

reserved

Examples: *inform_int := DV_STATE(dev_name_str)*

11.45 EOF Built-In Function

The EOF built-in function returns a Boolean value based on a check to see if the end of file was encountered during the last read operation. The lun must be opened with “rwa” or “r” attributes, otherwise the end-of-file is not met and EOF always returns false.

Calling Sequence: *EOF (lun_id)*

Return Type: BOOLEAN

Parameters: *lun_id* : INTEGER

[IN]

Comments: *lun_id* can be any user-defined variable that represents an open logical unit number (LUN).

TRUE will be returned if the last operation on *lun_id* was a READ and the end of file was reached before all specified variables were read.

If the end of file is reached before the READ statement completed, all variables not yet read are set to uninitialized.

If *lun_id* is associated to a communication port, the end-of-file is not recognized, and in this case it is necessary to define in the program which character is assumed as file terminator and check when this character is encountered.

Examples:

```

OPEN FILE lun_id ('data.txt', 'R')
READ lun_id (str)
WHILE NOT EOF (lun_id)
    WRITE (str, NL)
    READ lun_id (str)
ENDWHILE

OPEN FILE lun_id2 ('COM0:', r) WITH $FL_NUM_CHARS = 1
bool_var := FALSE
WHILE NOT bool_var DO
    READ lun_id2 (str::1)
    IF str = '\033' THEN -- assume '!' as end of file
        WRITE ('Found EOF', NL)
        CLOSE FILE lun_id2
    ENDIF
ENDWHILE

```

11.46 ERR_POST Built-In Procedure

The ERR_POST built-in procedure can be used by Users for raising an error in similar way the System does. Which type of error that is raised is defined by the supplied arguments.

Active alarms are included; they require responses by the User. When the user has selected the response, the program is notified (signal event or variable set) and optionally the Active alarm is removed. Active alarms can also be removed programmatically using the CANCEL ALARM statement and providing the alarm identifier.

When ERR_POST is executed, the error is treated like any error generated by the System.

Calling Sequence: `ERR_POST (error_num, error_str, error_sev <, post_flags> <, resp_array> <, alarm_id> <, resp_data>)`

Parameters:

error_num : INTEGER	[IN]
error_str : STRING	[IN]
error_sev : INTEGER	[IN]
post_flags : INTEGER	[IN]
resp_array : ARRAY [4] OF INTEGER	[IN]
alarm_id : INTEGER	[OUT]
resp_data : INTEGER	[OUT]

Comments: `error_num` can be any INTEGER expression whose value is in the range of 43008 to 44031. Such errors correspond to EC_USR1 and EC_USR2 class.

`error_str` is a STRING expression that contains an error message to be displayed.

`error_sev` is an INTEGER expression whose value is in the range of 2 to 10. The severity specified in the `ERR_POST` call determines the effects on the state of the system. Available error severities:

- 2 : Warning
- 4 : Pause, hold if holdable
- 6 : Pause all, hold if holdable
- 8 : Hold
- 10 : DRIVE OFF

The error is generally logged and can be monitored using error events (e.g. WHEN ERRORNUM BY) inside CONDITIONS.

Note that \$ERROR variable is NOT set for the program that calls the `ERR_POST` built-in. For testing if the error specified in the `ERR_POST` occurred, the \$SYS_ERROR predefined variable can be used.

`post_flags` is an INTEGER expression whose value consists of the following masks:

- Bit 1: Do not log the error in the binary log file (.lbe)
- Bit 2: Do not display the error on the status bar of the TP screen
- Bit 3: Do not trigger any enabled error event CONDITIONS
- Bit 4: Do not display the error on the scrolling window of the TP system screen
- Bit 5: Do not display the error on the scrolling window of WinC5G Terminal
- Bit 6: Do not cause effects on the general state of the System (based on error severity)
- Bit 7: If this bit is set, the program state will have transitions based on error severity
- Bit 10: It is typically useful when the program execution is inside a routine. If set, it causes the execution to immediately return from the called routine (bit 7 must also be set).



It is recommended not to set bit 10 when severity is 2: due to the fact that, upon `ERROR_POST` execution, the control is returned to the calling program, this could cause the routine to be infinitely called again until a different situation occurs.

- Bit 13: Log the error in the binary log file (.lbe) even if the error is being posted with severity 2
- Bit 14: Do not report on the TP
- Bit 15: Generates a latched alarm. This will require an operation from the user for unlatching the alarm itself, for example from the Alarm Latched page or UtilityLogLatchAcknowledge command from TP-INT.
- Bit 16: Generates a latched alarm which disables the possibility of DriveOn until such an alarm is reset by the User
- Bit 17: Generates a latched alarm which disables the possibility of switching the System to AUTO (LOCAL) state until such an alarm is reset by the User

- Bit 18 : Do not set as an Active alarm
- Bit 19 : Do not add to the log of messages
- Bit 21 : A process alarm - displayed in a different colour
- Bit 22 : An alarm that cannot be reset. After this you cannot remove the alarm state.
- Bit 23 : Informational alarm
- Bit 27 : Add to the action log file (as opposed to error log file)

resp_array : this parameter is an ARRAY OF INTEGER of 4 elements. The meaning of each array element is described here below:

- [1] : **response mask** - this is a mask which represents the different responses (e.g. Ok, Cancel, etc.) that the User can select. These responses will be shown on the menu of the TP Alarm page. Each response can be defined referring to an ERR_xx predefined constant. Maximum number of allowed responses is 5. An example of setting this response mask field is : (1 SHL ERR_OK - 1) OR (1 SHL ERR_CANCEL - 1) for showing OK and CANCEL in the bottom menu.
- [2] : **Default response**
 - 0 - Don't do any action
 - -1 - Cancel the alarm but do not perform any of the actions
 - >0 - Do the alarm response specified in element [3]
- [3] : **response action** - after the user has acknowledged the alarm, it is possible to display an event or set a variable with the following meaning:
 - 0 - The alarm is cancelled in any case (this is the default)
 - 1 - An event, specified in element [4], is raised and the variable, defined in *resp_data* parameter, is set to the value explained below
 - 2 - A variable, defined in the *resp_data*, is set to the value explained below.
 - 16 - Error is program specific, so if the program is deactivated the alarm is reset
 - 32 - Allow only one single response to the alarm.
- [4] : **response action code** - this is the event code to be raised when the User acknowledges the Active alarm. The event code must be a valid User event code. This is the same as SIGNAL EVENT i.e. 49152 to 49663

alarm_id : optional variable (by reference) that will contain the alarm identifier after the alarm has become active

resp_data : optional integer variable whose value is set to zero when the alarm is raised and when the alarm is acknowledged the low word is the *alarm_id* and the high word the user's response. The high word is zero if the alarm is reset and no user response. Please note that it is not possible to delete this variable from memory until the error is acknowledged.

Examples:

```

PROGRAM actalarm NOHOLD
VAR vi_id, vi_data : INTEGER
    wi_conf : ARRAY[4] OF INTEGER
BEGIN
    CYCLE
    --Configure items to be displayed in the bottom menu when alarm occurs
    wi_conf[1] := (1 SHL ERR_OK - 1) OR (1 SHL ERR_CANCEL - 1)
    OR (1 SHL ERR_RETRY - 1) OR (1 SHL ERR_ABORT - 1) OR (1 SHL
    ERR_SKIP - 1)
    wi_conf[2] := ERR_OK    -- Default
    wi_conf[3] := 1      -- Post an event
    wi_conf[4] := 49152   -- Event code
    vi_data := 0
    --Raise the error

```

```

ERR_POST(43009 + $CYCLE, 'Sample error', 4, 0, wi_conf, vi_id, vi_data)--Wait
for the user response
WAIT FOR EVENT 49152
WRITE LUN_CRT ('User responded. Id=', vi_data AND 0xFFFF::8::2, 'Response=',
vi_data SHR 16::8::2, NL)

--Now cancel the alarm
CANCEL ALARM vi_id
END actalarm

PROGRAM alarm NOHOLD
VAR wi_response_cfg : ARRAY[4] OF INTEGER
  vi_resp_alarm_id, vi_button, vi_resp_sel, vi_alarm_id : INTEGER
BEGIN
  -- To raise the event
  -- Configure the response button to be displayed (OK & CANCEL)
  wi_response_cfg[1] := (1 SHL ERR_OK - 1) + (1 SHL ERR_CANCEL - 1)
  -- Configure the default response (Perform the OK)
  wi_response_cfg[2] := ERR_OK
  -- Configure the response flags (ie. Event)
  wi_response_cfg[3] := 1
  -- Configure the response channel / code
  wi_response_cfg[4] := 49170

  -- Zero out the response flag
  vi_resp_sel := 0

  -- Raise the event
  ERR_POST(43009, 'No water', 2, 0, wi_response_cfg, vi_resp_sel, vi_alarm_id)

  -- Wait for the reply
  WAIT FOR EVENT 49170 OR (vi_resp_sel <> 0)
  -- Get the response alarm id
  vi_resp_alarm_id := vi_resp_sel AND 0xFFFF
  WRITE ('Alarm id;', vi_resp_alarm_id)
  vi_button := vi_resp_sel SHR 16
  SELECT vi_resp_sel OF
    CASE (ERR_OK):
      -- Do whatever
    CASE (ERR_CANCEL):
      -- Do whatever
  ENDSELECT

  CANCEL ALARM vi_resp_alarm_id
END alarm

```

11.47 ERR_STR Built-In Function

This built-in function returns the message string associated to a specific error.

Calling Sequence: str := ERR_STR(err_num<,err_sev,err_flags> <,language_int>)

Return Type: STRING

Parameters:	err_num: INTEGER	[IN]
	err_sev: INTEGER	[OUT]

<code>err_flags:INTEGER</code> <code>language_int: INTEGER</code>	<code>[OUT]</code> <code>[IN]</code>
--	---

Comments:

`err_num` is the error number whose message is looked for
`err_sev` is the error severity
`err_flags` are the flags for the returned string. If not specified, the string is returned as %s and no newline.
 0x0002 no first letter capitalization
 0x0004 no newline at the end.
`language_int` is the specified language:

- 0 = default
- 1 = English
- 2 = French
- 3 = German
- 4 = Italian
- 5 = Portuguese
- 6 = Spanish
- 7 = Turkish
- 8 = Chinese.

11.48 ERR_TRAP Built-In Function

This built-in function indicates whether the specified error number does really exist or not.

Calling Sequence: `exist := ERR_TRAP(err_num)`

Return Type: BOOLEAN

Parameters: `err_num: INTEGER` [IN]

Comments: `err_num` is the error number whose existence is to be checked

11.49 ERR_TRAP_OFF Built-In Procedure

The ERR_TRAP_OFF built-in procedure turns error trapping OFF for the specified errors.

Calling Sequence: `ERR_TRAP_OFF (error_num1<, error_num2<, .. error_num8>>)`

Parameters: `error_num1 .. error_num8: INTEGER` [IN]

Comments: `error_num1..error_num8` indicate the numbers of the errors to be trapped off. While error trapping is turned off, error checking is performed by the system. This is the normal case. The maximum number is 8; negative numbers are allowed to invert the ON/OFF.

Examples:

```

PROGRAM fib NOHOLD
VAR s : STRING[30]
ROUTINE filefound(as_name : STRING) : BOOLEAN EXPORTED FROM fib
ROUTINE filefound(as_name : STRING) : BOOLEAN
BEGIN
    ERR_TRAP_ON(39960) -- trap SYS_CALL errors
    SYS_CALL('fv', as_name)
    ERR_TRAP_OFF(39960)

```

```

IF $SYS_CALL_STS > 0 THEN -- check status of SYS_CALL
  RETURN(FALSE)
ELSE
  RETURN(TRUE)
ENDIF
END filefound
BEGIN
CYCLE
  WRITE ('Enter file name: ')
  READ (s)
  IF filefound(s) = TRUE THEN
    WRITE ('*** file found ***', NL)
  ELSE
    WRITE ('*** file NOT found ***', NL)
  ENDIF
END fib

```

See also: [ERR_TRAP_ON Built-In Procedure](#)

11.50 ERR_TRAP_ON Built-In Procedure

The ERR_TRAP_ON built-in procedure turns error trapping ON for the specified errors.

Calling Sequence: `ERR_TRAP_ON (error_num1<, error_num2<, .. error_num8>>)`

Parameters: `error_num : INTEGER` [IN]

Comments: `error_num1..error_num8` indicate the numbers of the errors to be trapped on. Only those errors in the EC_TRAP group (from 39937 to 40109) can be used.

The maximum number is 8; negative numbers are allowed to invert the ON/OFF.

While error trapping is turned on, the program is expected to handle the specified errors. \$ERROR or other status predefined variables (\$SYS_CALL_STS, etc.) will indicate the error that occurred even if the error is currently being trapped.

Note that to avoid that an error occurs for a program, in case of multiple ERR_TRAP_ON for the same error, it is useful to execute ERR_TRAP(39997).

Examples:

```

PROGRAM fib NOHOLD
VAR s : STRING[30]
ROUTINE filefound(as_name : STRING) : BOOLEAN EXPORTED FROM fib
ROUTINE filefound(as_name : STRING) : BOOLEAN
BEGIN
  ERR_TRAP_ON(39960) -- trap SYS_CALL errors
  SYS_CALL(fv, as_name)
  ERR_TRAP_OFF(39960)
  IF $SYS_CALL_STS > 0 THEN -- check status of SYS_CALL
    RETURN(FALSE)
  ELSE
    RETURN(TRUE)
  ENDIF
END filefound
BEGIN
  CYCLE
  WRITE ('Enter file name: ')
  READ (s)
  IF filefound(s) = TRUE THEN
    WRITE ('*** file found ***', NL)
  ELSE
    WRITE ('*** file not found ***', NL)
  ENDIF
END

```

```

        ENDIF
    END flib

```

See also: [ERR_TRAP_OFF Built-In Procedure](#)

11.51 EXP Built-In Function

The EXP built-in function returns the value of e, the base of the natural logarithm, raised to a specified power.

Calling Sequence: EXP (power)

Return Type: REAL

Parameters: power : REAL [IN]

Comments: The maximum value for *power* is 88.7.

e is the base of a natural logarithm, approximately 2.71828

Examples:

```

x := EXP(1)      -- x = 2.71828
x := EXP(15.2)   -- x = 3992786.835
x := EXP(4.1)    -- x = 60.34028

```

11.52 FL_BYT_ES_LEFT Built-In Function

The FL_BYT_ES_LEFT built-in function returns the number of bytes available at the specified logical unit number (LUN).

Calling Sequence: FL_BYT_ES_LEFT (lun_id)

Return Type: INTEGER

Parameters: lun_id : INTEGER [IN]

Comments: *lun_id* can be any user-defined variable that represents an open logical unit number (LUN).

lun_id must have been opened for random access and read or an error occurs.
End of line markers are counted in the returned value

Examples:

```

OPEN FILE file_lun ('data.txt', 'RWA'),
    WITH $FL_RANDOM = TRUE,
ENDOPEN
FL_SET_POS(file_lun, 0) -- beginning of the file
...
IF FL_BYT_ES_LEFT(file_lun) > 0 THEN
    get_next_value
ELSE
    write_error
ENDIF

```

11.53 FL_GET_POS Built-In Function

The FL_GET_POS built-in function returns the current file position for the next read/write operation.

Calling Sequence: FL_GET_POS (lun_id)

Return Type: INTEGER

Parameters: lun_id : INTEGER [IN]

Comments: *lun_id* can be any user-defined variable that represents an open logical unit number (LUN).
lun_id must have been opened for random access and read or an error results.
 End of line markers are counted in the returned value

Examples:

```

PROGRAM test NOHOLD
VAR i, lun : INTEGER
arr : ARRAY[20] OF REAL
ROUTINE write_values(values : ARRAY[*] OF REAL; lun : INTEGER)
VAR i : INTEGER
back_patch : INTEGER
total : INTEGER (0) -- initialize to 0
BEGIN
  back_patch := FL_GET_POS(lun)
  WRITE lun (0, NL)
  FOR i := 1 TO ARRAY_DIM1(values) DO
    IF values[i] <> 0 THEN
      total := total + 1
      WRITE lun (values[i], NL)
    ENDIF
  ENDFOR
  OPEN FILE lun ('temp.txt', 'rw'),
    WITH $FL_RANDOM = TRUE,
  ENDOPEN
  write_values(arr, lun)
  CLOSE FILE lun
END test
ELSE
  ENDIF
ENDIF
ENDFOR
-- backup and output number of values written to the file
FL_SET_POS(lun, back_patch)
WRITE lun (total)
FL_SET_POS(lun, 1) -- return to end of file
END write_values

```

11.54 FL_SET_POS Built-In Procedure

The FL_SET_POS built-in procedure sets the file position for the next read/write operation.

Calling Sequence: FL_SET_POS (lun_id, file_pos)

Parameters: lun_id : INTEGER [IN]
 file_pos : INTEGER [IN]

Comments: *file_pos* must be between -1 and the number of bytes in the file:
 -1 means the file position is to be set to the end of the file
 0 means the file position is to be set to the beginning of the file.
 Any other number represents an offset from the start of the file, in bytes, to which the file position is to be set.
 End of line markers should be counted in the *file_pos* value.

lun_id can be any user-defined INTEGER variable that represents an open logical unit number (LUN).

lun_id must be opened for random access and read or an error results.

Examples:

```

PROGRAM test NOHOLD
VAR i, lun : INTEGER
arr : ARRAY[20] OF REAL
ROUTINE write_values(values : ARRAY[*] OF REAL; lun : INTEGER)
VAR i : INTEGER
back_patch : INTEGER
total : INTEGER (0)      initialize to 0
BEGIN
    back_patch := FL_GET_POS(lun)
    WRITE lun (0, NL)

    FOR i := 1 TO ARRAY_DIM1(values) DO
        IF values[i] <> 0 THEN
            total := total + 1
            WRITE lun (values[i], NL)
        ELSE
        ENDIF
    ENDFOR

    -- backup and output number of values written to the file
    FL_SET_POS(lun, back_patch)
    WRITE lun (total)
    FL_SET_POS(lun, 1)      -- return to end of file
END write_values
BEGIN
    -- fill the array
    FOR i := 1 TO 20 DO
        arr[i] := i
    ENDFOR
    OPEN FILE lun ('temp.txt', 'rw'),
        WITH $FL_RANDOM = TRUE,
    ENDOPEN
    write_values(arr, lun)
    CLOSE FILE lun
END test

```

11.55 FL_STATE Built-In Function

The FL_STATE built-in function obtains information about a file.

Calling Sequence: `FL_STATE (file_name_str, < size_int <, time_int <, attr_int <, checksum <, properties >>>)`

Return Type: BOOLEAN

Parameters:	file_name_string : STRING	[IN]
	size_int : INTEGER	[OUT]
	time_int : INTEGER	[OUT]
	attr_int : INTEGER	[OUT]
	checksum : INTEGER	[OUT]
	properties : ARRAY [6] of STRING	[OUT]

Comments: The return value indicates whether the file exists or not.
`file_name_string` identifies the file whose information is to be obtained.
The filename, excluding the device, directory path and extension, must NOT exceed 32 characters.
The error can be trapped by `ERR_TRAP_ON(39990)` in order not to stop the program.
`size_int` will be set to the size of the file in bytes.
`time_int` will be set to the time of the file creation.
`attr_int` will be set to the attributes of the file, such as hidden, read_only, or system. The following table outlines the meaning of the bits of `attr_int`:

- Bit 1 : read_only file
- Bit 2 : hidden file
- Bit 3 : system file
- Bit 5: directory
- Bit 6: folder (.ZIP)

As an example, if a file is hidden and read_only, the value of `attr_int` will be 3 (bits 1 and 2 are set).
`checksum` will be set to the checksum of the file content
`properties` is an array of 6 strings, that will be set to the following information:

- [1] - username of the file creator (`$PROP_AUTHOR`)
- [2] - file creation time (`$PROP_DATE`)
- [3] - software version (`$PROP_VERSION`)
- [4] - host computer on which the file has been created (`$PROP_HOST`)
- [5] - release (`$PROP_REVISION`)
- [6] - title (`$PROP_TITLE`)

Examples:

```
exist_bool := FL_STATE(file_name_str)
exist_bool := FL_STATE(file_name_str, size_int, time_int)
```

11.56 FLOW_MOD_ON Built-In Procedure

This routine can be used for enabling the Flow modulation algorithm.

Calling Sequence: `FLOW_MOD_ON (port, flow_tbl_idx)`

Parameters:

<code>port</code> :	INTEGER	[IN]
<code>flow_tbl_idx</code> :	INTEGER	[IN]

Comments: `port` is an INTEGER port reference (`$AOUT[]`, `$FMO[]`, `$WORD[]`).
`flow_tbl_idx`: is the index of the `$FLOW_TBL` that contains all the settings to be used during the algorithm application.

For disabling the algorithm `FLOW_MOD_OFF` built-in procedure must be called.
Please note that, if the program which called `FLOW_MOD_ON` is deactivated, the algorithm is automatically disabled.

reference to motion programming manual for further details about this algorithm.

See also: [FLOW_MOD_OFF Built-In Procedure](#)

11.57 FLOW_MOD_OFF Built-In Procedure

This routine can be used for disabling the Flow modulation algorithm.

Only the program which issued FLOW_MOD_ON can disable the algorithm by calling current routine.

Calling Sequence: FLOW_MOD_ON (flow_tbl_idx)

Parameters: flow_tbl_idx: INTEGER [IN]

Comments: *flow_tbl_idx*: is the index of the \$FLOW_TBL that contains all the settings to be used during the algorithm application.

Reference to Motion Programming manual for further details about this algorithm.

See also: [FLOW_MOD_ON Built-In Procedure](#)

11.58 HDIN_READ Built-In Procedure

The HDIN_READ built-in procedure reads the positional value saved (captured) upon a high speed digital input (\$HDIN) trigger.

Calling Sequence: HDIN_READ (pos_class <, channel_input, arm_num>)

Parameters: pos_class : || POSITION | JOINTPOS | XTNDPOS || [OUT]
channel_input : INTEGER [IN]
arm_num : INTEGER [IN]

Comments: *pos_class* indicates a POSITION, JOINTPOS, or XTNDPOS variable.
pos_class is set to the saved positional value.
channel_input indicates the channel (1 or 2) from which reading the captured positional quotes; the default value is 1. Obviously, the channel indicated by [HDIN_SET Built-In Procedure](#) and HDIN_READ Built-In Procedure, must be consistent!

arm_num is an optional arm number. If not specified, the default arm is used.
An HDIN_READ built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples: HDIN_READ(p1, 2, 1)

11.59 HDIN_SET Built-In Procedure

The HDIN_SET Built-In procedure is used to enable capturing the Arm quotes (positional values), setting a \$HDIN: High speed digital input.

Calling Sequence: HDIN_SET (arm, enbl, cont, record_flag <, channel_input, arm_num>...)

Parameters:	arm: INTEGER	[IN]
	enbl: BOOLEAN	[IN]
	cont: BOOLEAN	[IN]
	record_flag: BOOLEAN	[IN]
	channel_input : INTEGER	[IN]
	arm_num: INTEGER	[IN]

Comments: When a high speed digital input is set, the Arm can be stopped and its current position recorded.

arm indicates the Arm on which enabling capture.

enbl is a flag indicating whether the HDIN interrupt is enabled or disabled.

cont is a flag indicating whether the HDIN is continuously enabled or disabled after the first transition.

record_flag is a flag indicating whether the position is recorded upon HDIN. If it is recorded, HDIN_READ can be used to obtain the recorded value.

channel_input indicates channel 1 or 2 on which the high speed input is physically connected; the default channel is 1.

arm_num (is an Arm number or a sequence of numbers) indicates on which Arm(s), if any, the automatic LOCK function and an event are to be performed when a capture operation is triggered. Up to four arm numbers can be specified as separate parameters. If no Arms are specified, neither automatic LOCK functions nor events are enabled.

Note that even if this parameter is optional, the programmer is strongly recommended to specify it always, otherwise the transition events are not issued!

The HDIN interrupt is triggered by a transition (either positive to negative or negative to positive). For detecting the transition of the HDIN, events from 209 to 216 can be used in a CONDITION statement. Refer to [Condition Handlers](#) chapter for further details.

The HDIN interrupt is not triggered, also if enabled, if the predefined variable [\\$HDIN_SUSP: temporarily disables the use of \\$HDIN](#) has the value of 1. It is important to underline however that when HDIN_SET enables the HDIN interrupt, this predefined variable is zeroed, independently from its current value. Refer also to [par. 12.152 \\$HDIN: High speed digital input on page 435](#)

Examples: HDIN_SET(1, TRUE, FALSE, TRUE, 2, 1)

Program for HDIN handling;

The workpiece has been previously taught and the points have been stored in wp_siding;
The HDIN is used for a path search of the workpiece and, if triggered, all movements on
that piece are shifted.

```

PROGRAM hdin_ex DETACH
CONST          ki_siding = 6
VAR pnt0001j, pnt0002j, pnt0003j, pnt0004j: JOINTPOS FOR ARM[1]
    wp_siding: ARRAY[ki_siding] OF POSITION -- array of points
-- that determine the workpiece area
ROUTINE isr_siding
VAR lp_shift: POSITION
    lp_hdin_pos: POSITION -- position read upon HDIN
-- triggering
    lv_diff: VECTOR
    lv_i: INTEGER
BEGIN
    -- read the position of the robot upon a $HDIN transition

    HDIN_READ(lp_hdin_pos, 1, 1)
    -- get the offset between the first location of the workpiece
    -- and the position where the HDIN triggered
    lv_diff := POS_GET_LOC(lp_hdin_pos) - POS_GET_LOC
    (wp_siding[1])
    -- disable the HDIN reading
    HDIN_SET (1, FALSE, FALSE, FALSE, 1, 1)
    FOR lv_i :=1 TO ki_siding DO
        -- shift all motions on the workpiece by the difference
        -- previously calculated offsets
        lp_shift := wp_siding[lv_i]
        POS_SHIFT(lp_shift, lv_diff)
        MOVE LINEAR TO lp_shift
    ENDFOR
END isr_siding
BEGIN
CONDITION[1] NODISABLE:
    WHEN HOLD DO      -- temporarily disable the HDIN triggering
-- when the system goes in HOLD state
    $CRNT_DATA[1].HDIN_SUSP[1]:=1
    WHEN START DO     -- reenable the HDIN triggering when motion
-- restarts (START button)

```

```

$ CRNT _DATA[1] .HDIN_SUSP[1] := 0
ENDCONDITION
CONDITION [2] :
  WHEN EVENT 209 DO -- triggering of HDIN (negative transition)
    CANCEL ALL
    UNLOCK
    RESUME
    DISABLE CONDITION[1]
    isr_siding
  ENDCONDITION
CONDITION[3] : -- enable the HDIN when the motion starts
  WHEN AT START DO
    $ CRNT _DATA[1] .HDIN_SUSP[1] :=0
    ENABLE CONDITION[1], CONDITION[2]
  ENDCONDITION
  MOVE TO pnt0001j
CYCLE
RESUME -- in case the arm is locked
MOVE TO pnt0002j
-- setup the HDIN to lock the arm and record the position
HDIN_SET(1, TRUE, FALSE, TRUE, 1, 1)
$ CRNT _DATA[1] .HDIN_SUSP[1] := 1
-- pnt0004j is the first point of research of the workpiece
MOVE LINEAR TO pnt0004j WITH CONDITION[3]
...
END hdin_ex

```

11.60 IP_TO_STR Built-in Function

This function converts an integer in a string with the form of an IP address notation.

Calling Sequence: str_val := IP_TO_STR (int_val)

Return Type: STRING

Parameters: int_val : INTEGER [IN]
str_val : STRING [OUT]

Comments:

Examples: str_val := IP_TO_STR (0x200005A) -- str_val is set to
-- "90.0.0.2" value

11.61 IR_SET Built-In Procedure

This routine associates a bit of an INTEGER port (e.g. \$WORD) to an Interference Region.

Calling Sequence: IR_SET (port, bit_idx, ir_idx, signal_purpose<, b_val>)

Parameters: port: INTEGER [IN]
bit_idx: INTEGER [IN]
ir_idx: INTEGER [IN]
signal_purpose [IN]
b_val: BOOLEAN [IN]

Comments:	<p><i>port</i> is a digital output port associated to the being defined Interference Region <i>bit_idx</i> is the bit index within the <i>port</i> <i>ir_idx</i> is the index of the being defined Interference Region <i>signal_purpose</i> represents the meaning of the values of the output bit specified by the first three parameters of IR_SET built-in procedure.</p> <ul style="list-style-type: none"> - IR_PRESENCE - the bit indicates that the robot is inside the Region - IR_RESERVATION - the bit indicates that the robot has asked the permission for entering the region - IR_CONSENT - the bit indicates that the robot has the consent for entering the region. <p><i>b_val</i> is the boolean value (ON/OFF) the digital port bit is set to, whenever the Interference Region is entered. The default value is OFF.</p> <p>Refer to Motion Programming manual for further details about Interference Region optional feature.</p>
See also:	IR_SET_DIG Built-In Procedure IR_SWITCH Built-In Procedure
Examples:	<pre>-- Definition of Interference Region no.1, as hybrid \$IR_TBL[1].IR_TYPE := IR_HYBRID -- Definition of the sphere with center in pnt0001P -- position, radius=250mm, used for Arm 1, IR no.1 IR_CreateSphere(pnt0001P, 250, 1, 1) -- Associate IR no.1 to \$FMO[1] flexible output, bit 1, for -- presence information IR_SET(\$FMO[1], 1, 1, IR_PRESENCE) -- Associate IR no.1 to \$FMO[1] flexible output, bit 2, for -- reservation information IR_SET(\$FMO[1], 2, 1, IR_RESERVATION) -- Associate IR no.1 to \$FMI[1] flexible input, bit 1, for -- the consent IR_SET(\$FMI[1], 1, 1, IR_CONSENT) -- Activation of Interference Region no.1 IR_SWITCH(ON, 1)</pre>

11.62 IR_SET_DIG Built-In Procedure

This routine associates a boolean digital port to an Interference Region .

Calling Sequence:	IR_SET_DIG (digital_port, ir_idx, signal_purpose<, b_val>)
Parameters:	<p><i>digital_port</i>: INTEGER [IN] <i>ir_idx</i>: INTEGER [IN] <i>signal_purpose</i>: INTEGER [IN] <i>b_val</i>: BOOLEAN [IN]</p>
Comments:	<p><i>digital_port</i> is a digital output port associated to the being defined Interference Region <i>ir_idx</i> is the index of the being defined Interference Region <i>signal_purpose</i> represents the meaning of the values of the output port specified by the first three parameters of IR_SET built-in procedure.</p> <ul style="list-style-type: none"> - IR_PRESENCE - the bit indicates that the robot is inside the Region - IR_RESERVATION - the bit indicates that the robot has asked the permission for entering the region

- IR_CONSENT - the bit indicates that the robot has the consent for entering the region.

b_val is the boolean value (ON/OFF) the digital port is set to, whenever the Interference Region is entered. The default value is OFF.

Refer to **Motion Programming** manual for further details about Interference Region optional feature.

See also:

[IR_SET Built-In Procedure](#)
[IR_SWITCH Built-In Procedure](#)

Examples:

```
-- Definition of Interference Region no.1, as hybrid
$IR_TBL[1].IR_TYPE := IR_HYBRID

-- Definition of the sphere with center in pnt0001P
-- position, radius=250mm, used for Arm 1, IR no.1
IR_CreateSphere(pnt0001P, 250, 1, 1)

-- Associate IR no.1 to $DOUT[1] digital output for
-- presence information
IR_SET_DIG($DOUT[1], 1, IR_PRESENCE)
-- Associate IR no.1 to $DOUT[2] digital output for
-- reservation information
IR_SET_DIG($DOUT[2], 1, IR_RESERVATION)
-- Associate IR no.1 to $DIN[1] digital input for
-- the consent
IR_SET_DIG($DIN[1], 1, IR_CONSENT)

-- Activation of Interference Region no.1
IR_SWITCH(ON, 1)
```

11.63 IR_SWITCH Built-In Procedure

This routine activates/deactivates the specified Interference Region .

Calling Sequence: `IR_SWITCH (act_flag, ir_idx)`

Parameters: `act_flag: BOOLEAN` [IN]
`ir_idx: INTEGER` [IN]

Comments: `act_flag` indicates whether activate or deactivate the specified Interference Region

`ir_idx` is the index of the being activated/deactivated Interference Region

Refer to **Motion Programming** manual for further details about Interference Region optional feature.

See also:

[IR_SET Built-In Procedure](#)

11.64 IS_FLY Built-In Function

The IS_FLY built-in function returns a boolean value (TRUE, FALSE) that indicates if the MOVE statement, eventually being interpreted by the specified program, is a MOVEFLY or a normal MOVE. If it is a MOVEFLY it returns TRUE, otherwise it returns FALSE.

Note that IS_FLY only checks the MOVE that is being interpreted and not the one already being executed by the robot. This function is useful for understanding the kind of move (fly or not) from a function that is in WITH clause with the motion statement.

Calling Sequence: IS_FLY (prog_name)

Return Type: BOOLEAN

Parameters: prog_name : STRING [IN]

Comments: prog_name is the name of the program that is interpreting the MOVE to be checked.

Examples:

```

PROGRAM prg_1
VAR p1: position
ROUTINE f1 (spd: INTEGER) : INTEGER
VAR b: BOOLEAN

```

```

BEGIN
    b := IS_FLY('prg_1')
    IF B = TRUE THEN -- case of MOVEFLY
        -- undertakes the foreseen statements in case of FLY
    ELSE
        -- undertakes the foreseen statements in case of no fly
    ENDIF
END f1
BEGIN
    MOVEFLY TO p1 ADVANCE WITH $PAR = f1(10)
END prg_1

```

11.65 JNT Built-In Procedure

This function returns a JOINTPOS composed by the specified location.

Calling Sequence: JNT (*jnt_var*, <*ax1,ax2,ax3,ax4,ax5,ax6,ax7,ax8,ax9,ax10*>)

Parameters:	<i>jnt_var</i> : JOINTPOS	[OUT]
	<i>ax1</i> : REAL	[IN]
	<i>ax2</i> : REAL	[IN]
	<i>ax3</i> : REAL	[IN]
	<i>ax4</i> : REAL	[IN]
	<i>ax5</i> : REAL	[IN]
	<i>ax6</i> : REAL	[IN]
	<i>ax7</i> : REAL	[IN]
	<i>ax8</i> : REAL	[IN]
	<i>ax9</i> : REAL	[IN]
	<i>ax10</i> : REAL	[IN]

Comments: *jnt_var* is the destination var which the single axis will be assigned to.
ax1 to *ax10* are optional parameters containing the value of the joint to be assigned to the JOINTPOS.

The rules to be followed for using this built-in are listed here below:

- the number of real values specified must be the same as the last axis (not the total number of axes)
- if the axis does not exist, then the value specified must be zero
- if the error is trapped on, then a difference in number of axes is allowed.

Examples: JNT (*pnt0002j*, 10.2, 0, 0, 5.7)

```

Assume to have a machine of axis 1,2,3,4,5,6,-,8:
JNT(j1,1,2,3,4,5,6,0,7) -- allowed as all axes are specified
JNT(j1,1,2,3,4,5,6,1,7) -- ERROR AS AXIS 7 is being set even if
                           -- not existing
ERR_TRAP_ON(40065)
JNT(j1,10,20) -- allowed now, because it is trapped on
JNT(j1,10,20,30,40,50,60,70) -- allowed, but 70 is set for axis 7
                               -- even if not existing: this is
                               -- because the error is trapped on.

```

11.66 JNT_SET_TAR Built-In Procedure

The JNT_SET_TAR built-in procedure sets the value of the position of an axis.

Calling Sequence: JNT_SET_TAR(*axis*, *value* <, *arm_num*>)

Parameters:	<i>axis</i>	: INTEGER	[IN]
	<i>value</i>	: REAL	[IN]
	<i>arm_num</i>	: INTEGER	[IN]

Comments: *axis* indicates the axis to be modified.

value is the position value in degrees to assign to the axis.

arm_num is an optional parameter specifying the desired arm to modify.

If the specified axis is not present, or the axis is not enabled for this operation, or the value is not valid, an error will be detected.

In order to correctly execute the JNT_SET_TAR built-in, it is mandatory to have an integer number (for example 124) for the transmission rate (\$TX_RATE).

This means that, for each axis turn (360° of the axis), the motor (resolver)

undertakes an integer number of turns. This transmission rate cannot be an approximation of a real value because this could cause an unrecoverable error of axis positioning.

During the built-in execution, it is needed that all axes linked to the machine (Servo CPU board) are steady, that there are no pending or active movements and that there are not positioning transient. It is suggested to use a well defined positioning range of tolerance (\$TERM_TYPE := FINE or \$TERM_TYPE := JNT_FINE) for the motion that precedes the JNT_SET_TAR call. It is a good rule, after having executed the built-in, to wait about one second before executing new movements in order to complete the process of position update.

Examples: PROGRAM *rotate*

```

CONST
  axis = 7
  arm_num = 1
VAR
  pnt0001x, pnt0002x, pnt0003x, pnt0004x : XTNDPOS
ROUTINE weld(sch : INTEGER) EXPORTED FROM weldrout
BEGIN
  MOVE JOINT TO pnt0001x -- Move rotating table to 30 Degree
  CYCLE
  WAIT FOR $DIN[10]          -- Wait for workpiece to be mounted
  MOVE JOINT TO pnt0002x -- Rotate table 180 degrees

```

```

weld(1)
MOVE JOINT TO pnt0003x -- Rotate table 120 degrees
weld(2)
MOVE JOINT TO pnt0004x -- Rotate table 60 degrees
weld(3)
-- Set the position of axis 7 to 30 degrees
JNT_SET_TAR(axis, 30, arm_num)
END rotate

```

11.67 JNTP_TO_POS Built-In Procedure

The JNTP_TO_POS built-in procedure converts a JOINTPOS expression to a POSITION variable. This conversion is performed using the current \$BASE, \$TOOL and \$UFRAME; however, when calling JNTP_TO_POS built-in procedure, it is possible to pass some optional parameters which are the reference frames to be used during the conversion.

Calling Sequence: JNTP_TO_POS (*jnt_expr*, *pos_var* <,*base_ref*, *tool_ref*, *ufr_ref*, *dyn_flg*>)

Parameters:	<i>jnt_expr</i> : JOINTPOS	[IN]
	<i>pos_var</i> : POSITION	[OUT]
	<i>base_ref</i> : POSITION	[IN]
	<i>tool_ref</i> : POSITION	[IN]
	<i>ufr_ref</i> : POSITION	[IN]
	<i>dyn_flg</i> : BOOLEAN	[IN]

Comments: *jnt_expr* is the JOINTPOS expression to be converted.
pos_var is the POSITION variable that gets set to the result of the conversion.
base_ref is the \$BASE to be used while converting to POSITION
tool_ref is the \$TOOL to be used while converting to POSITION
ufr_ref is the \$UFRAME to be used while converting to POSITION
dyn_flg is a flag to indicate whether or not dynamic references (such as conveyors, active cooperative axes/arms, etc.) should be used while converting to POSITION.

A JNTP_TO_POS built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples:

JNTP_TO_POS (<i>jp1</i> , <i>p1</i>)	-- converts from jointpos to position
<i>v1</i> := VEC(0, 100, 0)	-- creates vector
POS_SHIFT(<i>p1</i> , <i>v1</i>)	-- shifts position by vector
POS_TO_JNTP(<i>p1</i> , <i>jp1</i>)	-- converts shifted position to jointpos

See also: [POS_TO_JNTP Built-In Procedure](#)

11.68 KEY_LOCK Built-In Procedure

The KEY_LOCK built-in procedure locks out either the keyboard of the PC when acting on the command menu of the Terminal or the Teach Pendant keypad or both.

Calling Sequence: KEY_LOCK (*physical_device*, *enabl_disabl* <, *flags*>)

Parameters: physical_device : INTEGER [IN]
 enabl_disabl : BOOLEAN [IN]
 flags : INTEGER [IN]

Comments: *physical_device* is the device(s) to be locked out. The valid devices are PDV_TP and PDV_CRT. These should be ORed together when locking out both the PC keyboard and the Teach Pendant keypad.
enabl_disabl specifies whether the device(s) are to be enabled (TRUE) or disabled (FALSE).
flags is an optional parameter specifying when the device is to be automatically unlocked. The bits of the value indicate which system states should cause the automatic unlocking. Refer to the description of \$SYS_STATE in [Predefined Variables List](#) chapter for the possible values.
 An error occurs if the EXECUTE, EDIT, or TEACH environments are active or the state selector is set to REMOTE or AUTO.
 The device is automatically unlocked when the system enters the PROGR state, power failure recovery is initiated, or a fatal error is detected.

Examples: KEY_LOCK(PDV_TP, FALSE) -- disables the TP keyboard
 KEY_LOCK(PDV_CRT, FALSE) -- disables the PC keyboard
 KEY_LOCK(PDV_TP, TRUE) -- enables the TP keyboard

11.69 LN Built-In Function

The LN built-in function returns the natural logarithm of a number.

Calling Sequence: LN (number)

Return Type: REAL

Parameters: number : REAL [IN]

Comments: number must be greater than zero or an error results.

Examples: X := LN(5) -- x = 1.61
 X := LN(62.4) -- x = 4.13356
 X := LN(angle - offset)

11.70 MEM_SPACE Built-In Procedure

MEM_SPACE built-in procedure returns information about space in C5G memory.

Calling Sequence: MEM_SPACE (total, num_blocks, largest)

Parameters: total : INTEGER [OUT]
 num_blocks : INTEGER [OUT]
 largest : INTEGER [OUT]

Comments: *total* is the total number of free bytes available.
num_blocks is the number of individual free blocks.
largest is the size, in bytes, of the largest block available.

Examples: MEM_SPACE(total, num_blocks, largest)
 WRITE('Total free bytes: ', total, NL)
 WRITE('Number of free blocks: ', num_blocks, NL)
 WRITE('Largest available block: ', largest, NL)

11.71 NODE_APP Built-In Procedure

NODE_APP built-in procedure appends uninitialized nodes to the end of a PATH variable.

Calling Sequence:	NODE_APP (path_var <, num_nodes>)	
Parameters:	path_var : PATH	[IN, OUT]
	num_nodes : INTEGER	[IN]
Comments:	<p><i>path_var</i> is the PATH variable to be modified. <i>num_nodes</i> is the number of nodes to be appended to the path. If not specified, one node is appended. The user-defined fields and the destination standard node fields (\$MAIN_ and/or \$AUX_) of the appended nodes will be uninitialized. The non-destination standard node fields will be set to the same values as the last node in the path. A NODE_APP built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.</p>	
Examples:	<pre>NODE_APP(weld_pth) -- appends one node NODE_APP(weld_pth, 4) -- appends four nodes</pre>	

11.72 NODE_DEL Built-In Procedure

The NODE_DEL built-in procedure deletes nodes from a PATH variable.

Calling Sequence:	NODE_DEL (path_var, node_idx <, num_nodes>)	
Parameters:	path_var : PATH	[IN, OUT]
	node_idx : INTEGER	[IN]
	num_nodes : INTEGER	[IN]
Comments:	<p><i>path_var</i> is the PATH variable to be modified. <i>node_idx</i> is the index number of the first node to be deleted. It must be in the range from 1 to the total number of nodes in the path. <i>num_nodes</i> is the number of nodes to delete from the path, starting with <i>node_idx</i>. If not specified, one node is deleted. If this built-in is called on a path having no nodes, an error will be detected. After a node is deleted, all following nodes are renumbered. If a deleted node had a symbolic name, that symbolic name can be reused for a different node.A NODE_DEL built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.</p>	
Examples:	<pre>ROUTINE clear_path (name : PATH OF weld_node) BEGIN NODE_DEL(name, 1, PATHLEN(name)) END clear_path</pre>	

11.73 NODE_GET_NAME Built-In Procedure

The NODE_GET_NAME built-in procedure obtains the symbolic name of a path node.

Calling Sequence:	NODE_GET_NAME (path_var, node_num, name)
--------------------------	--

Parameters: path_var : PATH [IN]
 node_num : INTEGER [IN]
 name : STRING [OUT]

Comments: *path_var* is the PATH variable for the node.
 The node name *node_num* is obtained.
name will be set to the symbolic name of *node_num* node of *path_var* path. If the symbolic name is longer than the maximum length of *name*, it will be truncated.
 An error occurs if node *node_num* does not have a symbolic name.
 An error occurs if *node_num* is less than 1 or greater than the current length of *path_var*.
 A NODE_GET_NAME built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples: NODE_GET_NAME (weld_pth, 3, str_var)

11.74 NODE_INS Built-In Procedure

The NODE_INS built-in procedure inserts uninitialized nodes into a PATH variable.

Calling Sequence: NODE_INS (path_var, node_idx <, num_nodes>)

Parameters: path_var : PATH [IN, OUT]
 node_idx : INTEGER [IN]
 num_nodes : INTEGER [IN]

Comments: *path_var* is the PATH variable to be modified.
 The new nodes will be inserted after node *node_idx*. *node_idx* must be in the range from 0 to the total number of nodes in the path.
num_nodes is the number of nodes to insert into the path. If not specified, one node is inserted.
 All nodes from *node_idx+1* to the end of the path are renumbered.
 The user-defined fields and the destination standard node fields (\$MAIN_ and/or \$AUX_) will be uninitialized. The non-destination standard node fields will be set to the same values as *node_idx* node.
 A NODE_INS built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples: NODE_INS (weld_pth, 3) -- inserts one node after node 3
 NODE_INS (weld_pth, 3, 4) -- inserts four nodes after node 3

11.75 NODE_SET_NAME Built-In Procedure

The NODE_SET_NAME built-in procedure assigns or clears the symbolic name of a path node.

Calling Sequence: NODE_SET_NAME (path_var, node_num <, name>)

Parameters: path_var : PATH [IN]
 node_num : INTEGER [IN]
 name : STRING [IN]

Comments: *path_var* is the PATH variable for the node.
node_num is the node whose symbolic name is either being set or cleared.
name is the new symbolic name for node.

It is an error if the name is already used for a different node in *path_var* or if it is an empty string ("").

If name is not specified, the symbolic name of the node is cleared.

An error occurs if *node_num* is less than 1 or greater than the current length of *path_var*.

A NODE_SET_NAME built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples: `NODE_SET_NAME (weld_pth, 3, start_welding)`

11.76 ON_JNT_SET Built-In Procedure

The ON_JNT_SET built-in procedure, used in case of On Pos feature, associates a bit of an INTEGER port (e.g. \$WORD) to the state of the arm in respect with the reaching of the position defined in the predefined variable \$ON_POS_TBL[on_pos_idx].OP_JNT.

When this position is reached, the bit is set to 1 and the \$ON_POS_TBL[on_pos_idx].OP_REACHED assumes the value of TRUE. For making active this association it is however necessary to call the ON_POS (ON...) on the same \$ON_POS_TBL element. The ON_JNT_SET should be chosen instead of ON_POS_SET when auxiliary axes are present and shall be checked. Note that the value of \$ON_POS_TBL[on_pos_idx].OP_JNT variable cannot be directly assigned as the variable is read-only; one of the parameters to this procedure is a JOINTPOS and its value will be assigned to \$ON_POS_TBL[on_pos_idx].OP_JNT field at the moment of the calling.

Calling Sequence: `ON_JNT_SET (port, bit, on_pos_idx, jnt_pos <, jnt_mask>)`

Parameters:	port : INTEGER	[IN]
	bit : INTEGER	[IN]
	on_pos_idx : INTEGER	[IN]
	jnt_pos : JOINTPOS	[IN]
	jnt_mask : INTEGER	[IN]

Comments:
port is an INTEGER port reference (\$WORD, \$FMO[]).
bit is an INTEGER expression indicating the bit representing the \$ON_POS_TBL[on_pos_idx].OP_REACHED value.
on_pos_idx is the \$ON_POS_TBL index of the element associated to the port bit.

jnt_pos is assigned to the \$ON_POS_TBL[on_pos_idx].OP_JNT field, which is a PDL2 read-only variable. The arm of *jnt_pos* shall be the same as the arm passed to the ON_POS(ON, *on_pos_idx<,arm_num>*) that has to be called next.

jnt_mask is the mask of the joints that are checked during the robot motion for determining if the \$ON_POS_TBL[on_pos_idx].OP_JNT has been reached. If some joints or auxiliary axes have to be excluded from this checking, it is sufficient to pass the corresponding bit of this mask with the 0 value. *jnt_mask* is an optional parameter and, if not specified, it assumes the default value of \$ARM_DATA[arm_num].JNT_MASK.

Examples:

```
-- program for NH4 robot and integrated rail
PROGRAM op45
VAR p1, p2, p3 : POSITION
VAR j1, j2 : JOINTPOS
VAR i : INTEGER
```

```

BEGIN
  -- disable On Pos feature for the first 5 $ON_POS_TBL
  -- elements
  FOR i := 1 TO 5 DO
    ON_POS(FALSE, i)
  ENDFOR

  CONDITION[1] NODISABLE :
    WHEN EVENT 134 DO
      $FDOUT[21] := TRUE
    WHEN EVENT 135 DO
      $FDOUT[22] := TRUE
    WHEN EVENT 136 DO
      $FDOUT[23] := TRUE
  ENDCONDITON
  CONDITION[2] NODISABLE :
    WHEN EVENT 142 DO
      $FDOUT[21] := FALSE
    WHEN EVENT 143 DO
      $FDOUT[22] := FALSE
    WHEN EVENT 144 DO
      $FDOUT[23] := FALSE
  ENDCONDITON
  ENABLE CONDITION[1], CONDITION[2]
  $TOOL_RMT := TRUE
  $BASE := POS(0, 0, 0, 0, 0, 0, '')
  $TOOL := POS(1000, 3000, -1000, 180, 0, 0, '')
  $UFRAME := POS(0, 0, 50, 0, 90, 0, '')
  -- On Pos definition on POSITIONs for the first 3
elements
  -- of $ON_POS_TBL
  ON_POS_SET($WORD[20], 1, 1)    -- element 1 uses bit 1 of
  -- $WORD[20]
  ON_POS_SET($WORD[20], 2, 2)    -- element 2 uses bit 3 of
  -- $WORD[20]
  ON_POS_SET($WORD[20], 3, 3)    -- element 3 uses bit 3 of
  -- $WORD[20]
  -- On Pos definition on JOINTPOS
  ON_JNT_SET($WORD[20], 4, 4, j1, $JNT_MASK)    -- element 4
  -- uses bit 4 of $WORD[20]
  ON_JNT_SET($WORD[20], 5, 5, j2, 0x40)    -- element 4 uses
  -- bit 4 of $WORD[20] FOR i := 1 TO 5 DO
    $ON_POS_TBL[i].OP_TOOL := $TOOL
    $ON_POS_TBL[i].OP_UFRAME := $UFRAME
    $ON_POS_TBL[i].OP_TOOL_DSBL := FALSE
    $ON_POS_TBL[i].OP_TOOL_RMT := TRUE
  ENDFOR
  $ON_POS_TBL[1].OP_POS := p1
  $ON_POS_TBL[2].OP_POS := p2
  $ON_POS_TBL[3].OP_POS := p3
  -- Enable On Pos feature for the first 5 $ON_POS_TBL
  -- elements
  FOR i := 1 TO 5 DO
    ON_POS(TRUE, i, 1)
  ENDFOR

  -- Associate bit 8 of $WORD [20] to the On

```

```

-- Trajectory feature
ON_TRAJ_SET($WORD[20], 8, 1)
-- Start of Cycle loop
CYCLE

MOVE TO p1
DELAY 1000
MOVE TO p2
DELAY 1000
MOVE LINEAR TO p3
DELAY 1000
MOVE TO j1
DELAY 1000

MOVE LINEAR TO j2
DELAY 1000
END op45

```

See also:

[ON_POS Built-In Procedure](#), [ON_POS_SET Built-In Procedure](#),
On Position Feature in Motion Programming Manual and
[\\$ON_POS_TBL: ON POS table data](#).

11.77 ON_JNT_SET_DIG Built-In Procedure

This built-in procedure, used in case of On Pos feature, associates a digital port (e.g. \$DOUT) to the state of the arm in respect with the reaching of the position defined in the predefined variable \$ON_POS_TBL[on_pos_idx].OP_JNT.

When this position is reached, the port is set to ON and the \$ON_POS_TBL[on_pos_idx].OP_REACHED assumes the value of TRUE.

For making this association active it is needed to call the ON_POS (ON, ...) procedure on the same \$ON_POS_TBL element. \$OP_TOOL and \$OP_UFRAME fields of \$ON_POS_TBL must be initialized in addition to \$OP_JNT.

ON_JNT_SET_DIG should be chosen instead of ON_POS_SET_DIG when auxiliary axes are present and shall be checked. Note that the value of \$ON_POS_TBL[on_pos_idx].OP_JNT variable cannot be directly assigned as the variable is read-only; one of the parameters to this procedure is a JOINTPOS and it its value will be assigned to \$ON_POS_TBL[on_pos_idx].OP_JNT field at the moment of the calling.

Its use is very similar to ON_JNT_SET. The difference is only related to the port data type: ON_JNT_SET refers to an INTEGER port while the ON_JNT_SET_DIG refers to a digital (BOOLEAN) port.

Calling Sequence:	ON_JNT_SET_DIG (port, on_pos_idx, jnt_pos <, jnt_mask>)												
Parameters:	<table border="0"> <tr> <td>port</td><td>: BOOLEAN</td><td>[IN]</td></tr> <tr> <td>on_pos_idx</td><td>: INTEGER</td><td>[IN]</td></tr> <tr> <td>jnt_pos</td><td>: JOINTPOS</td><td>[IN]</td></tr> <tr> <td>jnt_mask</td><td>: INTEGER</td><td>[IN]</td></tr> </table>	port	: BOOLEAN	[IN]	on_pos_idx	: INTEGER	[IN]	jnt_pos	: JOINTPOS	[IN]	jnt_mask	: INTEGER	[IN]
port	: BOOLEAN	[IN]											
on_pos_idx	: INTEGER	[IN]											
jnt_pos	: JOINTPOS	[IN]											
jnt_mask	: INTEGER	[IN]											
Comments:	<p><i>port</i> is a BOOLEAN port reference (\$DOUT, \$BIT, \$OUT, \$WORD_BIT, \$FMO_BIT).</p> <p><i>on_pos_idx</i> is the \$ON_POS_TBL index of the element associated to the digital port.</p> <p><i>jnt_pos</i> is assigned to the \$ON_POS_TBL[on_pos_idx].OP_JNT field. The arm of this position must match with the arm to the ON_POS built-in that has to be called next.</p>												

jnt_mask is the mask of the joints that are checked during the robot motion for determining if the \$ON_POS_TBL[on_pos_idx].OP_JNT has been reached. To exclude some axis from being checked, it is sufficient to pass the corresponding bit of this mask with the 0 value. *jnt_mask* is an optional parameter and, if not specified, it assumes the default value of \$ARM_DATA[arm_num].JNT_MASK.

Examples: See the example of ON_JNT_SET.

See also: [ON_POS Built-In Procedure](#), [ON_JNT_SET Built-In Procedure](#) and [\\$ON_POS_TBL: ON POS table data](#).

11.78 ON_POS Built-In Procedure

The ON_POS built-in procedure allows the enabling and the disabling of the On Pos feature using the values assigned in one element of the \$ON_POS_TBL table.

Calling Sequence: ON_POS (flag, on_pos_idx<, arm_num>)

Parameters:

flag	: BOOLEAN	[IN]
on_pos_idx	: INTEGER	[IN]
arm_num	: INTEGER	[IN]

Comments: *flag* is a BOOLEAN value indicating if the algorithm should be enabled (ON) or disabled (OFF).

on_pos_idx is the \$ON_POS_TBL index of the element to which the ON POS feature should apply.

arm_num is an optional parameter containing the arm number. If not specified, the default arm is used. This parameter is only consider upon ON_POS(ON).

Since the ON_POS (ON, ...) execution, the predefined variables

\$ON_POS_TBL[on_pos_idx].OP_REACHED and the port bit defined with the ON_POS_SET are respectively assigned to TRUE and 1 when the \$ON_POS_TBL[on_pos_idx].OP_POS or \$ON_POS_TBL[on_pos_idx].OP_JNT are reached. Any condition event (WHEN EVENT 134..149) enabled for this table element could trigger.

Note that the ON_POS call should be preceeded by the setting of all fields of the \$ON_POS_TBL element and by the calling to the ON_POS_SET or ON_JNT_SET built-in procedure.

After the ON_POS (ON, ...), the On Pos feature remains enabled until the next ON_POS (OFF, ...) on the same \$ON_POS_TBL element or the next controller restart. Note that, upon ON_POS (OFF, ...) the related bit assumes the value of 0).

Examples: PROGRAM op44

```

VAR p1, p2, p3, p4 : POSITION
VAR i : INTEGER
BEGIN
-- definition of condition that monitor the entering and
-- the exiting from the sphere defined in the $ON_POS_TBL.
  CONDITION[1] NODISABLE:
    WHEN EVENT 134 DO
      $FDOUT[21] := TRUE
    WHEN EVENT 135 DO
      $FDOUT[22] := TRUE
    WHEN EVENT 136 DO
      $FDOUT[23] := TRUE
  ENDCONDITION
  CONDITION[2] NODISABLE :
    WHEN EVENT 142 DO

```

```

        $FDOUT[21] := FALSE
WHEN EVENT 143 DO
        $FDOUT[22] := FALSE
WHEN EVENT 144 DO
        $FDOUT[23] := FALSE
ENDCONDITION
ENABLE CONDITION[1], CONDITION[2]
-- definition of bits 1, 2, 3 of $WORD[5]
-- associated to element 1,2,3 of the $ON_POS_TBL
ON_POS_SET($WORD[5], 1, 1); ON_POS_SET($WORD[5], 2, 2)
ON_POS_SET($WORD[5], 3, 3)
FOR i := 1 TO 3 DO
    $ON_POS_TBL[i].OP_TOOL := $TOOL
    $ON_POS_TBL[i].OP_UFRAME := $UFRAME
    $ON_POS_TBL[i].OP_TOOL_DSBL:= FALSE
    $ON_POS_TBL[i].OP_TOOL_RMT := FALSE
ENDFOR
-- Home Position
$ON_POS_TBL[1].OP_POS := p1
-- Enabling of ON POS feature on element 1 of the
-- $ON_POS_TBL
ON_POS(TRUE, 1, 1)

-- Tip dress position
$ON_POS_TBL[2].OP_POS := p2
ON_POS(TRUE, 2, 1)
-- Service position
$ON_POS_TBL[3].OP_POS := p3
ON_POS(TRUE, 3, 1)
CYCLE
MOVE ARM[1] TO p1; MOVE ARM[1] TO p2; MOVE ARM[1] TO p3
END op44
    
```

See also:

[ON_POS_SET Built-In Procedure](#),
[\\$ON_POS_TBL: ON POS table data](#) and
 On Position Feature in **Motion Programming** manual.

11.79 ON_POS_SET Built-In Procedure

This built-in procedure allows, in case of ON POS feature, to associate a bit of an INTEGER port (e.g. \$WORD) to the state of the arm in respect with the reaching of a position defined in the predefined variable \$ON_POS_TBL[on_pos_idx].OP_POS. When this position is reached, the bit assumes the value of 1 and the \$ON_POS_TBL[on_pos_idx].OP_REACHED assumes the value of TRUE. For making active this association it is however necessary to call the ON_POS (ON,..) on the same \$ON_POS_TBL element.

The \$OP_TOOL and \$OP_UFRAME fields of \$ON_POS_TBL must be initialized in addition to \$OP_POS.

Calling Sequence: `ON_POS_SET (port, bit, on_pos_idx)`

Parameters:	port : INTEGER	[IN]
	bit : INTEGER	[IN]
	on_pos_idx : INTEGER	[IN]

Comments:	<i>port</i> is an INTEGER port reference (e.g. \$WORD). <i>bit</i> is an INTEGER expression indicating the bit representing the \$ON_POS_TBL[on_pos_idx].OP_REACHED value. <i>on_pos_idx</i> is the \$ON_POS_TBL index of the element associated to the port bit.
Examples:	See example associated to ON_POS Built-In Procedure
See also:	ON_POS_SET_DIG Built-In Procedure , ON_POS Built-In Procedure , \$ON_POS_TBL: ON POS table data and On Position Feature in Motion Programming manual

11.80 ON_POS_SET_DIG Built-In Procedure

This built-in procedure allows, in case of ON POS feature, to associate a digital port (e.g. \$DOUT) to the state of the arm in respect with the reaching of a position defined in the predefined variable \$ON_POS_TBL[on_pos_idx].OP_POS. When this position is reached, the port is set to ON and the \$ON_POS_TBL[on_pos_idx].OP_REACHED assumes the value of TRUE. For making active this association it is needed to call the ON_POS (ON,...) on the same \$ON_POS_TBL element. The \$OP_TOOL and \$OP_UFRAME fields of \$ON_POS_TBL must be initialized in addition to \$OP_POS. Its use is very similar to ON_POS_SET one. The difference is only related to the port data type: ON_POS_SET refers to an INTEGER port while ON_POS_SET_DIG refers to a digital (BOOLEAN) port.

Calling Sequence:	ON_POS_SET_DIG (port, on_pos_idx)						
Parameters:	<table border="0"> <tr> <td>port</td> <td>: BOOLEAN</td> <td>[IN]</td> </tr> <tr> <td>on_pos_idx</td> <td>: INTEGER</td> <td>[IN]</td> </tr> </table>	port	: BOOLEAN	[IN]	on_pos_idx	: INTEGER	[IN]
port	: BOOLEAN	[IN]					
on_pos_idx	: INTEGER	[IN]					
Comments:	<i>port</i> is a BOOLEAN port reference (e.g. \$DOUT). <i>on_pos_idx</i> is the \$ON_POS_TBL index of the element associated to the port bit.						
Examples:	ON_POS_SET_DIG (\$DOUT[5], 3)						
See also:	ON_POS_SET Built-In Procedure , ON_POS Built-In Procedure , \$ON_POS_TBL: ON POS table data and On Position Feature in Motion Programming manual.						

11.81 ON_TRAJ_SET Built-In Procedure

The ON_TRAJ_SET built-in procedure defines a bit of an INTEGER port (e.g. \$WORD) to indicate if the robot is on the planned trajectory or not. When \$CRNT_DATA[arm_num].OT_COARSE assumes the TRUE value, this bit assumes the value of 1.

Calling Sequence:	ON_TRAJ_SET (port, bit <, arm_num>)									
Parameters:	<table border="0"> <tr> <td>port</td> <td>: INTEGER</td> <td>[IN]</td> </tr> <tr> <td>bit</td> <td>: INTEGER</td> <td>[IN]</td> </tr> <tr> <td>arm_num</td> <td>: INTEGER</td> <td>[IN]</td> </tr> </table>	port	: INTEGER	[IN]	bit	: INTEGER	[IN]	arm_num	: INTEGER	[IN]
port	: INTEGER	[IN]								
bit	: INTEGER	[IN]								
arm_num	: INTEGER	[IN]								
Comments:	<i>port</i> is an INTEGER port reference (e.g. \$WORD). <i>bit</i> is an INTEGER expression indicating the bit associated to the \$CRNT_DATA[arm_num].OT_COARSE value. <i>arm_num</i> is the number of the arm. If not specified, the default arm is used.									

Examples: The following example associates bit 3 of \$WORD[5] to the state of \$CRNT_DATA[2].OT_COARSE. When this variable value is TRUE, the bit has the value of 1.

```
ON_TRAJ_SET($WORD[5], 3, 2)
```

See also: Class of predefined variables having \$OT_ prefix in [Predefined Variables List](#) chapter,
[\\$CRNT_DATA: Current Arm data](#) fields and
 On Trajectory Feature in [Motion Programming](#) manual

11.82 ON_TRAJ_SET_DIG Built-In Procedure

This built-in procedure defines a BOOLEAN port (e.g. \$DOUT) to indicate if the robot is on the planned trajectory or not. When \$CRNT_DATA[arm_num].OT_COARSE assumes the TRUE value, this port is set to ON.

Calling Sequence: ON_TRAJ_SET_DIG (port, bit <, arm_num>)

Parameters: port : BOOLEAN [IN]
 arm_num : INTEGER [IN]

Comments: *port* is a BOOLEAN port reference (e.g. \$DOUT).
arm_num is the number of the arm. If not specified, the default arm is used.

Examples: ON_TRAJ_SET_DIG (\$DOUT[7], 1)

See also: Class of predefined variables having \$OT_ prefix in [Predefined Variables List](#) chapter,
[ON_TRAJ_SET Built-In Procedure](#) and
 On Trajectory Feature in [Motion Programming](#) manual

11.83 ORD Built-In Function

ORD built-in function returns the numeric ASCII code of a character in a STRING.

Calling Sequence: ORD (src_string, index)

Return Type: INTEGER

Parameters: src_string : STRING [IN]
 index : INTEGER [IN]

Comments: *src_string* is the STRING containing the character.
index is the position in *str* of the character whose ASCII value is to be returned.
 If *index* is less than one or greater than the current length of *src_string*, an error occurs.

Examples: PROGRAM ordinal

```

VAR
    src_string : STRING[10]
    X : INTEGER
BEGIN
    src_string := ABCDEF
    X := 0
    X := ORD(src_string, 7)    -- ERROR: index out of range
    X := ORD(src_string, -1)   -- ERROR: index out of range
    X := ORD(src_string, 2)    -- X now equals 66 (ASCII for B)
END ordinal

```

```

PROGRAM e NOHOLD
VAR s : STRING[20] NOSAVE
VAR vi_value, vi_j, vi_len : INTEGER NOSAVE
BEGIN
  s := '\199A\127\129\000N'
  vi_len := STR_LEN(s)
  FOR vi_j := 1 TO vi_len DO
    vi_value := ORD(s, vi_j)
    IF vi_value < 0 THEN
      vi_value += 256 -- Correct the value
    ENDIF
    WRITE LUN_CRT ('Index ', vi_j, ' Value: ',
    vi_value:3:-5, NL)
  ENDFOR
END e

```

11.84 PATH_GET_NODE Built-In Procedure

The PATH_GET_NODE built-in procedure obtains the node number of a PATH variable corresponding to a given symbolic name.

Calling Sequence: PATH_GET_NODE (path_var, name, node_num)

Parameters:	path_var	: PATH	[IN]
	name	: STRING	[IN]
	node_num	: INTEGER	[OUT]

Comments:
path_var is the PATH variable for the node.
name is the symbolic name to be searched for.
node_num will be set to the node number corresponding to the symbolic name.
 It is an error if *name* is not a symbolic name of any nodes in *path_var*.
 A PATH_GET_NODE built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples:
 NODE_SET_NAME(*weld_pth*, 3, *start_welding*)
 NODE_INS(*weld_pth*, 1, 2) -- insert two nodes after node 1
 PATH_GET_NODE(*weld_pth*, *start_welding*, *i*) -- *i* = 5

11.85 PATH_LEN Built-In Function

The PATH_LEN built-in function returns the length of a PATH variable. This indicates the number of nodes currently in the PATH.

Calling Sequence: PATH_LEN (path_var)

Return Type: INTEGER

Parameters:	path_var : PATH	[IN]
--------------------	-----------------	------

Comments:
 The returned value is the number of nodes currently in *path_var*.
 Zero is returned if *path_var* is uninitialized.

Examples:
 ROUTINE *clear_path* (*name* : PATH OF *weld_node*)
 BEGIN
 NODE_DEL(*name*, 1, PATHLEN(*name*))
 END *clear_path*

11.86 POS Built-In Function

The POS built-in function returns a POSITION composed of the specified location components, Euler angles, and configuration.

Calling Sequence: POS (x, y, z, e1, e2, e3, cnfg)

Return Type: POSITION

Parameters:	x : REAL	[IN]
	y : REAL	[IN]
	z : REAL	[IN]
	e1 : REAL	[IN]
	e2 : REAL	[IN]
	e3 : REAL	[IN]
	cnfg : STRING	[IN]

Comments: x, y, and z specify the Cartesian coordinates of a position.

e1, e2, and e3 specify the Euler angles of a position.

cnfg specifies the configuration string of a position. Refer to [Data Representation](#) chapter for a description of the configuration string.

Examples: ROUTINE shift_x (curpos : POSITION; shift_val : REAL)

```

VAR
    x, y, z, e1, e2, e3 : REAL
    cnfg : STRING[15]
BEGIN
    POS_XTRT(curpos, x, y, z, e1, e2, e3, cnfg)
    x := x + shift_val
    curpos := POS(x, y, z, e1, e2, e3, cnfg)
END shift_x

```

11.87 POS_COMP_IDL Built-In Procedure

The POS_COMP_IDL built-in procedure converts the compensated machine position value (that takes into account the offsets determined by the mechanical peculiarities of the machine) to the ideal one.

Calling Sequence: POS_COMP_IDL (pos_comp, pos_idl)

Parameters:	pos_comp : POSITION JOINTPOS XTNDPOS	[IN]
	pos_idl : POSITION JOINTPOS XTNDPOS	[OUT]

Comments: pos_comp is the compensated position.

pos_idl is the ideal position that is returned in output after the procedure call.

Examples: PROGRAM test

```

VAR p1, p2: POSITION
    j1, j2: JOINTPOS FOR ARM[1]
    x1, x2: XTNDPOS FOR ARM[1]
    x3, x4: XTNDPOS FOR ARM[2]
BEGIN
    POS_COMP_IDL (p1, p2)
    POS_COMP_IDL (j1, j2)
    POS_COMP_IDL (x1, x2)
    POS_COMP_IDL (x1, x3) -- error because the arm numbers
                           -- do not match

```

```

    POS_COMP_IDL (p1, j2) -- error because the data types do
                           -- not match
  END test

```

See also: [POS_IDL_COMP Built-In Procedure](#)

11.88 POS_CORRECTION Built-In Procedure

It is a functionality which allows to determine a robot POSITION, giving the angular variations X, Y and Z, in degrees. The final result is a position referred to the UFRAME (input datum: degrees around XYZ; output datum: new position expressed by means of Euler Angles).

Calling Sequence: `POS_CORRECTION (initial_pos, final_pos, angleX, angleY, angleZ, frame <, arm>)`

Parameters:	initial_pos	: POSITION	[IN]
	final_pos	: POSITION	[OUT]
	angleX	: REAL	[IN]
	angleY	: REAL	[IN]
	angleZ	: REAL	[IN]
	frame	: INTEGER	[IN]
	arm	: INTEGER	[IN]

Comments:
initial_pos is the initial position
final_pos is the calculated final position
angleX is the angular variation around X
angleY is the angular variation around Y
angleZ is the angular variation around Z
frame is the frame specification (BASE, TOOL, UFRAME)
arm is the optional arm number.

11.89 POS_FRAME Built-In Function

The POS_FRAME built-in function returns the frame specified by three or four Cartesian positions.

Calling Sequence: `POS_FRAME (corner, x, xy <, orig>)`

Return Type: POSITION

Parameters:	corner	: POSITION	[IN]
	x	: POSITION	[IN]
	xy	: POSITION	[IN]
	orig	: POSITION	[IN]

Comments: *orig* is the origin of the new frame. If *orig* is not specified, then *corner* is used as the origin.
 The x-axis of the new frame is parallel to a line defined by the points *corner* and *x*.
 The xy-plane is parallel to a plane defined by the points *corner*, *x*, and *xy*. *xy* is on the positive half of the xy-plane.
 The y-axis is on the xy-plane and is perpendicular to the x-axis.
 The z-axis is perpendicular to the xy-plane and intersects both the x- and y-axes. The positive direction on the z-axis is found using the right hand rule.
 This function is mainly useful for the definition of the user frame transformation \$UFRAME.

Examples: `$UFRAME := POS_FRAME(origin, xaxis, xyplane)`

11.90 POS_GET_APPR Built-In Function

The POS_GET_APPR built-in function returns the approach vector of the frame of reference specified by a position.

Calling Sequence: `POS_GET_APPR (source_pos)`

Return Type: VECTOR

Parameters: `source_pos : POSITION` [IN]

Comments: The approach vector represents the positive z direction of the frame of reference defined by *source_pos*.

Examples: `ROUTINE rotate_orient (posn : POSITION)`

```
VAR
    temp : VECTOR
BEGIN
    temp := POS_GET_APPR(posn)
    POS_SET_APPR(posn, POS_GET_NORM(posn))
    POS_SET_NORM(posn, temp)
END rotate_orient
```

11.91 POS_GET_CNFG Built-In Function

The POS_GET_CNFG built-in function returns the configuration string of a specified POSITION variable.

Calling Sequence: `POS_GET_CNFG (source_pos)`

Return Type: STRING

Parameters: `source_pos : POSITION` [IN]

Comments: The maximum length of a configuration string is 33. Therefore, if the return value of POS_GET_CNFG is assigned to a string variable, that variable should have a maximum length of at least 33 characters to avoid truncation.

Refer to [Data Representation](#) chapter for a description of the configuration string.

Examples: `cfg_str := POS_GET_CNFG(cur_pos)`

11.92 POS_GET_LOC Built-In Function

The POS_GET_LOC built-in function returns the location vector of a specified position.

Calling Sequence: `POS_GET_LOC (source_pos)`

Return Type: VECTOR

Parameters: `source_pos : POSITION` [IN]

Comments: The returned location vector represents the x, y, z components of *source_pos*.

Examples:

```

ROUTINE shift_loc (posn:POSITION; x_delta, y_delta, z_delta:REAL)
VAR
    alter, old, new : VECTOR
BEGIN
    alter := VEC(x_delta, y_delta, z_delta)
    old := POS_GET_LOC(posn)
    new := old # alter
    POS_SET_LOC(posn, new)
END shift_loc

```

11.93 POS_GET_NORM Built-In Function

The POS_GET_NORM built-in function returns the normal vector of the frame of reference specified by a position.

Calling Sequence: POS_GET_NORM (source_pos)

Return Type: VECTOR

Parameters: source_pos : POSITION [IN]

Comments: The normal vector represents the positive x axis of the frame of reference specified by a source_pos.

Examples: ROUTINE rotate_approach (posn : POSITION)

```

VAR temp : VECTOR
BEGIN
    temp := POS_GET_NORM(posn)
    POS_SET_NORM(posn, POS_GET_ORNT(posn) )
    POS_SET_ORNT(posn, temp)
END rotate_approach

```

11.94 POS_GET_ORNT Built-In Function

The POS_GET_ORNT built-in function returns the orientation vector of the frame of reference specified by a position.

Calling Sequence: POS_GET_ORNT (source_pos)

Return Type: VECTOR

Parameters: source_pos : POSITION [IN]

Comments: The orientation vector represents the positive y axis of the frame of reference defined by source_pos.

Examples: ROUTINE rotate_approach (posn : POSITION)

```

VAR temp : VECTOR
BEGIN
    temp := POS_GET_NORM(posn)
    POS_SET_NORM(posn, POS_GET_ORNT(posn) )
    POS_SET_ORNT(posn, temp)
END rotate_approach

```

11.95 POS_GET_RPY Built-In Procedure

This built-in procedure converts the angular coordinates of a position, expressed as

Euler Angle (Z, Y, Z), to three angles in RPY notation.

Calling Sequence: POS_GET_RPY (source_pos, roll, pitch, yaw)

Parameters:	source_pos : POSITION	[IN]
	roll : REAL	[OUT]
	pitch : REAL	[OUT]
	yaw : REAL	[OUT]

Comments: source_pos are the coordinates of the source position
roll is the rotation around Z axis
pitch is the rotation around Y axis
yaw is the rotation around X axis.

11.96 POS_IDL_COMP Built-In Procedure

The POS_IDL_COMP built-in procedure converts the ideal position of the machine into the compensated value, which takes into account the offsets determined by the mechanical peculiarities of the machine.

Calling Sequence: POS_IDL_COMP (pos_idl, pos_comp)

Parameters:	pos_idl : POSITION JOINTPOS XTNDPOS	[IN]
	pos_comp : POSITION JOINTPOS XTNDPOS	[OUT]

Comments: pos_idl is the ideal position
pos_comp is the compensated position that is returned in output after the procedure call.

Examples:

```

PROGRAM test
VAR p1, p2: POSITION
    j1, j2: JOINTPOS FOR ARM[1]
    x1, x2: XTNDPOS FOR ARM[1]
    x3, x4: XTNDPOS FOR ARM[2]
BEGIN
    POS_IDL_COMP (p1, p2)
    POS_IDL_COMP (j1, j2)
    POS_IDL_COMP (x1, x2)
    POS_IDL_COMP (x1, x3) -- error because the arm numbers
                           -- do not match
    POS_IDL_COMP (p1, j2) -- error because the data types do
                           -- not match
END test

```

See also: [POS_COMP_IDL Built-In Procedure](#)

11.97 POS_IN_RANGE Built-In Procedure

The POS_IN_RANGE built-in procedure sets a BOOLEAN value indicating whether a POSITION, JOINTPOS, or XTNDPOS value is in the range of a specified arm. This test is performed using the current \$BASE, \$TOOL and \$UFRAME; however, when calling POS_IN_RANGE built-in procedure, it is possible to pass some optional parameters which are the reference frames to be used while testing.

Calling Sequence: POS_IN_RANGE (test_pos, bool_ans <, arm_num> <, base_ref, tool_ref, ufr_ref, dyn_flg>)

Parameters:

test_pos	:	POSITION JOINTPOS XTNDPOS	[IN]
bool_ans	:	BOOLEAN	[OUT]
arm_num	:	INTEGER	[IN]
base_ref	:	POSITION	[IN]
tool_ref	:	POSITION	[IN]
ufr_ref	:	POSITION	[IN]
dyn_flg	:	BOOLEAN	[IN]

Comments:

test_pos is the position to be tested, to know whether it is in the range of the specified Arm.

bool_ans is set to TRUE if the arm can reach *test_pos* without stroke-end errors or cartesian position out of range error; otherwise, it is set to FALSE.

If no optional parameters are passed, the current \$BASE and \$TOOL are applied to *test_pos*.

If *arm_num* is not specified, the default arm is used.

base_ref is the \$BASE to be used while testing

tool_ref is the \$TOOL to be used while testing

ufr_ref is the \$UFRAME to be used while testing

dyn_flg is a flag to indicate whether or not dynamic references (such as conveyors, active cooperative axes/arms, etc.) should be used while testing.

A POS_IN_RANGE built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

This built-in is implemented by converting a POSITION or XTNDPOS variable to JOINTPOS format. In the case of a controller that does not have this capability, the POS_IN_RANGE does not return an error, but sets *bool_ans* to FALSE.

The user should therefore test \$ERROR and \$THRD_ERROR as shown in the second example below to fully understand the meaning of a FALSE outcome (i.e. whether the position is truly out of range, or that the built-in could not be properly executed).

Examples:

```

POS_IN_RANGE (posn, bool)
IF bool THEN
    MOVE TO posn
ELSE
    out_of_range (posn)
ENDIF

$ERROR:=0
POS_IN_RANGE (p, boo)
IF boo THEN
    MOVE TO p
ELSE
    IF $ERROR = 40028 THEN -- inverse kinematic not
available
        -- cannot determine if the position was in range
    ELSE
        -- position not in range
    ENDIF

```

11.98 POS_INV Built-In Function

The POS_INV built-in function returns the inverse of a Cartesian position.

Calling Sequence: POS_INV (source_pos)

Return Type: POSITION

Parameters: source_pos : POSITION [IN]

Comments: The returned value is the inverse of source_pos.
The configuration of the returned value is that of source_pos.

Examples:

```

ROUTINE get_flange_frame (posn, flange_pos : POSITION)
BEGIN
    flange_pos := posn : POS_INV ($TOOL) -- flange_pos represent the
    -- position of the flange frame when the TCP is in posn.
END get_flange_frame

```

11.99 POS_MIR Built-In Function

The POS_MIR built-in function returns the mirror image of a position with respect to the xz-plane of another position.

Calling Sequence: POS_MIR (source, mplane, orient)

Return Type: POSITION

Parameters:

source	: POSITION	[IN]
mplane	: POSITION	[IN]
orient	: BOOLEAN	[IN]

Comments: The configuration string is mirrored by negating the turn number values. The shoulder, elbow, and wrist configuration flags are unchanged. The turn configuration flags are negated.
The orientation is related to the Eulerian angles. If orient is TRUE, the Euler angles (and therefore the orientation) are mirrored. If orient is FALSE, the angles and the orientation are unchanged.
A suggested technique in utilizing this built-in function is to teach the position mplane in the tool frame after aligning the xz tool plane to the plane that will be considered as the source.

Examples: new_pos := POS_MIR (old_pos, mplane, orntatn)

11.100 POS_SET_APPR Built-In Procedure

The POS_SET_APPR built-in procedure sets orientation components of a POSITION variable by assigning the approach vector (z direction) of the corresponding frame of reference.

Calling Sequence: POS_SET_APPR (posn, appr_vec)

Parameters:

posn	: POSITION	[IN, OUT]
appr_vec	: VECTOR	[IN]

Comments: posn is the POSITION variable. An error will occur if it is uninitialized.
The positive z direction of the frame of reference defined by posn will be set to the vector specified by appr_vec.

Examples:

```

ROUTINE rotate_orient (posn : POSITION)
VAR temp : VECTOR
BEGIN
    temp := POS_GET_APPR (posn)
    POS_SET_APPR (posn, POS_GET_NORM (posn))
    POS_SET_NORM (posn, temp)
END rotate_orient

```

11.101 POS_SET_CNFG Built-In Procedure

The POS_SET_CNFG built-in procedure sets the configuration string of a POSITION variable.

Calling Sequence: POS_SET_CNFG (posn, new_cnfg)

Parameters: posn : POSITION [IN, OUT]
new_cnfg : STRING [IN]

Comments: *posn* is the POSITION variable. An error will occur if it is uninitialized. The configuration string of *posn* will be set to the value specified by *new_cnfg*. Refer to [Data Representation](#) chapter for a description of the configuration string.

Examples: POS_SET_CNFG (posn, SW)

11.102 POS_SET_LOC Built-In Procedure

The POS_SET_LOC built-in procedure sets the location vector of a POSITION variable.

Calling Sequence: POS_SET_LOC (posn, new_loc)

Parameters: posn : POSITION [IN, OUT]
new_loc : VECTOR [IN]

Comments: *posn* is the POSITION variable. An error will occur if it is uninitialized. The location component (x, y, z) of *posn* will be set to the location vector specified by *new_loc*.

Examples:

```

ROUTINE shift_loc (posn:POSITION; x_delta, y_delta, z_delta:REAL)
  VAR alter, old, new : VECTOR
  BEGIN
    alter := VEC(x_delta, y_delta, z_delta)
    old := POS_GET_LOC(posn)
    new := old # alter
    POS_SET_LOC(posn, new)
  END shift_loc

```

11.103 POS_SET_NORM Built-In Procedure

The POS_SET_NORM built-in procedure sets orientation components of a POSITION variable by assigning the normal vector (x direction) of the corresponding frame of reference.

Calling Sequence: POS_SET_NORM (posn, new_norm)

Parameters: posn : POSITION [IN, OUT]
new_norm : VECTOR [IN]

Comments: *posn* is the POSITION variable. An error will occur if it is uninitialized. The positive x axis of the frame of reference defined by *posn* will be set to the vector specified by *new_norm*.

Examples:

```

ROUTINE rotate_approach (posn : POSITION)
VAR temp : VECTOR
BEGIN
    temp := POS_GET_NORM(posn)
    POS_SET_NORM(posn, POS_GET_ORNT(posn) )
    POS_SET_ORNT(posn, temp)
END rotate_approach

```

11.104 POS_SET_ORNT Built-In Procedure

The POS_SET_ORNT built-in procedure sets orientation components of a POSITION variable by assigning the orientation vector (y direction) of the corresponding frame of reference.

Calling Sequence: POS_SET_ORNT (posn, new_orient)

Parameters: posn : POSITION [IN, OUT]
new_orient : VECTOR [IN]

Comments: posn is the POSITION variable. An error will occur if it is uninitialized. The positive y direction of the frame of reference defined by posn will be set to the vector specified by new_orient.

Examples:

```

ROUTINE rotate_approach (posn : POSITION)
VAR temp : VECTOR
BEGIN
    temp := POS_GET_NORM(posn)
    POS_SET_NORM(posn, POS_GET_ORNT(posn) )
    POS_SET_ORNT(posn, temp)
END rotate_approach

```

11.105 POS_SET_RPY Built-In Procedure

This built-in procedure converts three angles in RPY notation, to the angular coordinate, expressed as Euler Angle Z, Y, Z of a position.

Calling Sequence: POS_SET_RPY (pos_var, roll, pitch, yaw)

Parameters: pos_var : POSITION [IN]
roll : REAL [IN]
pitch : REAL [IN]
yaw : REAL [IN]

Comments: pos_var is position to be converted to the Z, Y, Z angular coordinates
roll is the rotation around Z axis
pitch is the rotation around Y axis
yaw is the rotation around X axis.

11.106 POS_SHIFT Built-In Procedure

The POS_SHIFT built-in procedure shifts a Cartesian position by a specified VECTOR.

Calling Sequence: POS_SHIFT (posn, shf_vec)

Parameters: posn : POSITION [IN, OUT]
shf_vec : VECTOR [IN]

Comments: *posn* is the POSITION variable to be shifted. If it is uninitialized, an error will occur.
shf_vec is the vector by which to shift *posn*. Its components are added to the location components of *posn*.

 **NOTE that POS_SHIFT Built-in Procedure shifts a Cartesian position using the CURRENT tool and frame references; the tool and frame references written in the WITH clause are not used at all.**

Examples:

```
JNTP_TO_POS(jp1, p1)      -- converts from jointpos to position
v1 := VEC(0, 100, 0)       -- creates vector
POS_SHIFT(p1, v1)          -- shifts position by vector
POS_TO_JNTP(p1, jp1)       -- converts shifted position to
                           jointpos
```

11.107 POS_TO_JNTP Built-In Procedure

The POS_TO_JNTP built-in procedure converts a POSITION expression to a JOINTPOS variable. This conversion is performed using the current \$BASE, \$TOOL and \$UFRAME; however, when calling POS_TO_JNTP built-in procedure, it is possible to pass some optional parameters which are the reference frames to be used during the conversion.

Calling Sequence: POS_TO_JNTP (pos_expr, jnt_var <,base_ref, tool_ref, ufr_ref, dyn_flg>)

Parameters:

pos_expr : POSITION	[IN]
jnt_var : JOINTPOS	[OUT]
base_ref : POSITION	[IN]
tool_ref : POSITION	[IN]
ufr_ref : POSITION	[IN]
dyn_flg : BOOLEAN	[IN]

Comments: *pos_expr* is the POSITION expression to be converted.
jnt_var is the JOINTPOS variable that results from the conversion.
base_ref is the \$BASE to be used while converting to JOINTPOS
tool_ref is the \$TOOL to be used while converting to JOINTPOS
ufr_ref is the \$UFRAME to be used while converting to JOINTPOS
dyn_flg is a flag to indicate whether or not dynamic references (such as conveyors, active cooperative axes/arms, etc.) should be used while converting to JOINTPOS.

A POS_TO_JNTP built-in call is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

Examples:

```
JNTP_TO_POS(jp1) -- converts from jointpos to position
v1 := VEC(0, 100, 0) -- creates vector
POS_SHIFT(p1, v1) -- shifts position by vector
POS_TO_JNTP(p1, jp1) -- converts shifted position to jointpos
```

See also: [JNTP_TO_POS Built-In Procedure](#)

11.108 POS_XTRT Built-In Procedure

The POS_XTRT built-in procedure extracts the location components, Euler angles, and configuration string of a Cartesian position.

Calling Sequence:	POS_XTRT (posn, x, y, z, e1, e2, e3, cnfg)	
Parameters:	posn : POSITION	[IN]
	x : REAL	[OUT]
	y : REAL	[OUT]
	z : REAL	[OUT]
	e1 : REAL	[OUT]
	e2 : REAL	[OUT]
	e3 : REAL	[OUT]
	cnfg : STRING	[OUT]
Comments:	<p><i>posn</i> is the POSITION from which the components are to be extracted. The Cartesian coordinates of <i>posn</i> are assigned to <i>x</i>, <i>y</i>, and <i>z</i>. The Euler angles of <i>posn</i> are assigned to <i>e1</i>, <i>e2</i>, and <i>e3</i>. The configuration string of <i>posn</i> is assigned to <i>cnfg</i>.</p>	
Examples:	<pre>ROUTINE shift_x (curpos : POSITION; shift_val : REAL) VAR x, y, z, e1, e2, e3 : REAL cnfg : STRING[15] BEGIN POS_XTRT(curpos, x, y, z, e1, e2, e3, cnfg) x := x + shift_val CURPOS := POS(x, y, z, e1, e2, e3, cnfg) END shift_x</pre>	

11.109 PROG_STATE Built-In Function

The PROG_STATE built-in function returns the current state of a program, as well as the program name, the line number being executed, the name of the being executed routine and the owning program.

Calling Sequence:	status := PROG_STATE (prog_name<, line_num> <, rout_name> <, ext_prog_name>)	
Return Type:	INTEGER	
Parameters:	prog_name : STRING	[IN]
	line_num : INTEGER	[OUT]
	rout_name : STRING	[OUT]
	ext_prog_name : STRING	[OUT]

Comments: *prog_name* is the name of the program for which the current state is to be returned. An error will occur if the program does not exist.
 The returned value is an INTEGER mask indicating the program state. Only some of the bits in the mask will have meaning for the user, other bits have internal uses and are reserved. The user should filter the returned value from PROG_STATE, ANDing it with 0x7FFF.
 If the program is not active, all bits are set to 1. Other common values include the following:

- 0 : active and running
- <0 : not active
 - -1 : unknown program
 - -2 : loaded, but not active
- >0 : suspended for some reason; please NOTE THAT more than one bit could be set, in the mask, at the same time.
 - 2 : paused
 - 4 : ready state (i.e. held)
 - 6 : ready-paused
 - 64 : waiting for a READ completion
 - 128 : waiting on a WAIT FOR statement
 - 256 : motion currently in progress
 - 512 : SYS_CALL currently in progress
 - 1024 : DELAY currently in progress
 - 2048 : waiting on a WAIT statement
 - 4096 : PULSE currently in progress

line_num is the returned number of the being executed line
rout_name is the returned name of the being executed routine

ext_prog_name is the returned name of the program which owns the being executed routine.

Examples:

```

PROGRAM psex
VAR vi_state, vi_line : INTEGER NOSAVE
  vs_rout, vs_owner : STRING[32] NOSAVE
ROUTINE r1
BEGIN
  vi_state := PROG_STATE($PROG_NAME, vi_line, vs_rout, vs_owner)
  WRITE LUN_CRT ('State = ', vi_state, ' Line =', vi_line, ' Rout =', vs_owner, '->', vs_rout, NL)
END r1

BEGIN
  vi_state := PROG_STATE($PROG_NAME, vi_line, vs_rout, vs_owner)
  WRITE LUN_CRT ('State = ', vi_state, ' Line =', vi_line, ' Rout =', vs_owner, '->', vs_rout, NL)
  r1
END psex

```

11.110 RANDOM Built-in Function

This function returns an INTEGER random number.

If no parameters or parameter less than 0 is specified, the random number is between 0 and 99.

Calling Sequence: `random_num := RANDOM(<max_num>)`

Return Type: INTEGER

Parameters: max_num : INTEGER [IN]

Comments: *max_num* is an optional parameter that indicates the maximum limit for the generated number. If not specified, the limit range is between 0 and 99.

11.111 ROUND Built-In Function

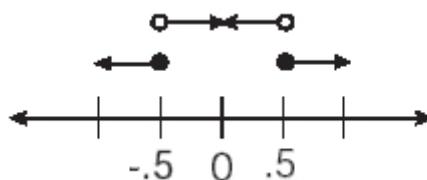
The ROUND built-in function rounds off a REAL number to return an INTEGER result.

Calling Sequence: ROUND (num)

Return Type: INTEGER

Parameters: num : REAL [IN]

Comments: The returned value is rounded down if *num* has a decimal value greater than 0.0 and less than 0.5. If the value is greater than or equal to 0.5 it will be rounded up (see diagram).
This function can be used to convert a REAL expression into an INTEGER value.



Examples:

ROUND(17.4)	-- result is 17
ROUND(96.5)	-- result is 97
ROUND(-17.4)	-- result is -17
ROUND(-96.5)	-- result is -97

11.112 SCRN_ADD Built-In Procedure

The SCRN_ADD built-in procedure adds a screen to a device so that it is included in the screen cycle of the SCRN key.

Calling Sequence: SCRN_ADD (dev_num, scrn_num)

Parameters: dev_num : INTEGER [IN]
scrn_num : INTEGER [IN]

Comments: The screen will be added to the screen cycle of the SCRN key on the device indicated by *dev_num*. The following predefined constants represent devices which are valid for *dev_num*:

- PDV_TP -- Teach Pendant
- PDV_CRT -- PC screen, when using WinC5G Program

scrn_num indicates the screen to be added. For user-created screens, this is the value obtained by the SCRN_CREATE built-in.
A screen is added to the cycle list of the SCRN key for a device only once no matter how many times the SCRN_ADD built-in is called.
An error occurs if *scrn_num* does not correspond to a valid screen.

Examples:

```

appl_screen := SCRN_CREATE(weld_disp, PDV_TP)
SCRN_ADD(PDV_TP, appl_screen)
IF SCRN_GET(PDV_TP) <> appl_screen THEN
    SCRN_SET(PDV_TP, appl_screen) -- want appl PC screen
ENDIF

```

11.113 SCRN_CLEAR Built-In Procedure

The SCRN_CLEAR built-in procedure clears all windows on the system screen or on the user-created screen.

Calling Sequence: SCRN_CLEAR (dev_num <, code <, scrn_num>>)

Parameters:

dev_num : INTEGER	[IN]
code : INTEGER	[IN]
scrn_num : INTEGER	[IN]

Comments: *dev_num* indicates the device containing the screen that is to be cleared. The following predefined constants represent devices that can be used for *dev_num*:
 PDV_TP -- Teach Pendant
 PDV_CRT -- PC screen (Terminal Window), when using WinC5G Program
code is an INTEGER expression indicating what to clear from the user screen.
 The following predefined constants can be used:

SCRN_CLR_CHR -- clear all characters on the specified screen
 SCRN_CLR_Rem -- remove all windows from the specified screen
 SCRN_CLR_DEL -- remove and delete all user created windows from the specified screen
 If *code* is not specified, SCRN_CLR_Rem is used as the default.
scrn_num indicates the user-created screen to be cleared. If not specified, the system created user screen (SCRN_USER) is cleared.

The SCRN_CREATE built-in function can be used to create new user screens.

Examples:

```

SCRN_CLEAR(PDV_CRT) -- Clears predefined user screen
-- on the PC
-- clear a user-created screen
SCRN_CLEAR(PDV_CRT, SCRN_CLR_CHR, appl_screen)
SCRN_CLEAR(PDV_TP) -- Clears predefined user screen
-- on the TP
SCRN_CLEAR(PDV_CRT, SCRN_CLR_DEL) -- windows are deleted

```

11.114 SCRN_CREATE Built-In Function

The SCRN_CREATE built-in function creates a new user screen.

Calling Sequence: SCRN_CREATE (scrn_name, dev_num)

Return Type: INTEGER

Parameters:

scrn_name : STRING	[IN]
dev_num : INTEGER	[IN]

Comments:	<p><i>scrn_name</i> is the name given to the new screen. <i>dev_num</i> indicates the device for which the screen is being created. The following predefined constants represent devices that can be used for <i>dev_num</i>:</p> <ul style="list-style-type: none"> PDV_TP -- Teach Pendant PDV_CRT -- PC screen (Terminal Window), when using WinC5G Program <p>The above devices can be combined using the OR operator in order to create a screen for both devices.</p> <p>The value returned will be a screen number which can be used in other built-in routines to indicate the newly created user screen.</p> <p>The new screen is not automatically added to the SCRN key cycle. Use the SCRN_ADD routine if the new screen should be cycled with the other screens. An error occurs if a screen called <i>scrn_name</i> already exists.</p>
Examples:	<pre>appl_screen := SCRN_CREATE(weld_disp, PDV_TP) SCRN_ADD(PDV_TP, appl_screen) IF SCRN_GET(PDV_TP) <> appl_screen THEN SCRN_SET(PDV_TP, appl_screen) -- want appl screen on TP ENDIF</pre>

11.115 SCRN_DEL Built-In Procedure

The SCRN_DEL built-in procedure deletes a user-created screen.

Calling Sequence:	SCRN_DEL (<i>scrn_name</i>)
Parameters:	<i>scrn_name</i> : STRING [IN]
Comments:	<p><i>scrn_name</i> indicates the screen to be deleted.</p> <p>The screen must be removed from any SCRN key cycles before deletion. Use the SCRN_REMOVE built-in for this purpose.</p> <p>An error occurs if <i>scrn_name</i> is not a defined screen or is currently in the cycle of a device SCRN key.</p>
Examples:	<pre>appl_screen := SCRN_CREATE(weld_disp, PDV_TP) SCRN_ADD(PDV_TP, appl_screen) IF SCRN_GET(PDV_TP) <> appl_screen THEN SCRN_SET(PDV_TP, appl_screen) -- want appl screen on TP ENDIF . . . SCRN_REMOVE(PDV_TP, appl_screen) SCRN_DEL(weld_disp);</pre>

11.116 SCRN_GET Built-In Function

The SCRN_GET built-in function returns the currently displayed screen on the specified device. In addition, the currently selected window on that screen can be obtained.

Calling Sequence:	SCRN_GET (<i>dev_num</i> <, <i>win_name</i> >)				
Return Type:	INTEGER				
Parameters:	<table border="0"> <tr> <td><i>dev_num</i> : INTEGER</td> <td>[IN]</td> </tr> <tr> <td><i>win_name</i> : STRING</td> <td>[OUT]</td> </tr> </table>	<i>dev_num</i> : INTEGER	[IN]	<i>win_name</i> : STRING	[OUT]
<i>dev_num</i> : INTEGER	[IN]				
<i>win_name</i> : STRING	[OUT]				

Comments: *dev_num* indicates the device for which the currently displayed screen is to be returned. The following predefined constants represent devices:
 PDV_TP -- Teach Pendant
 PDV_CRT -- PC screen (Terminal Window), when using WinC5G Program
 SCRN_USER -- user screen
 SCRN_SYS -- system screen
 SCRN_EDIT -- editor screenThe value returned will indicate a system created screen or a user-created screen. System created screens are represented by the following predefined constants:
 If the *win_name* parameter is specified, it will be set to the name of the window that is currently selected on the screen.

Examples:

```

IF SCRN_GET(PDV_TP) <> SCRN_SYS THEN
  SCRN_SET(PDV_TP, SCRN_SYS) -- want sys screen on TP
ENDIF
IF SCRN_GET(PDV_CRT) <> appl_screen THEN
  SCRN_SET(PDV_CRT, appl_screen) -- want appl screen on PC
ENDIF
  
```

11.117 SCRN_REMOVE Built-In Procedure

The SCRN_REMOVE built-in procedure removes a screen from a device so that it is no longer included in the screen cycle of the SCRN key.

Calling Sequence: SCRN_REMOVE (*dev_num*, *scrn_num*)

Parameters: *dev_num* : INTEGER [IN]
scrn_num : INTEGER [IN]

Comments: *dev_num* indicates the device to be used. The screen will be removed from the screen cycle of the SCRN key on that device. The following predef.constants represent devices:
 PDV_TP -- Teach Pendant
 PDV_CRT -- PC screen (Terminal Window), when running WinC5G Program

scrn_num indicates the user-created screen to be removed. This is the value obtained by the SCRN_CREATE built-in. An error occurs if *scrn_num* does not exist or is not currently in the cycle for the specified device.

Examples:

```

appl_screen := SCRN_CREATE(weld_disp, PDV_TP)
SCRN_ADD(PDV_TP, appl_screen)
IF SCRN_GET(PDV_TP) <> appl_screen THEN
  SCRN_SET(PDV_TP, appl_screen) -- want appl screen on TP
ENDIF
.
.
.
SCRN_REMOVE(PDV_TP, appl_screen)
SCRN_DEL(weld_disp);
  
```

11.118 SCRN_SET Built-In Procedure

The SCRN_SET built-in procedure forces the specified screen to be displayed on the specified device.

Calling Sequence: SCRN_SET (*dev_num*, *scrn_num*)

Parameters: *dev_num* : INTEGER [IN]
scrn_num : INTEGER [IN]

Comments:	<p><i>dev_num</i> indicates the device on which the specified screen is to be displayed. The following predefined constants represent devices: PDV_TP -- Teach Pendant PDV_CRT -- PC screen (Terminal Window), when running WinC5G Program <i>scrn_num</i> indicates the screen to be displayed. User-created screens or system created screens can be specified. System created screens are represented by the following predefined constants: SCRN_USER -- user screen SCRN_SYS -- system screen SCRN_EDIT -- editor screen If SCRN_EDIT is used for <i>scrn_num</i> and the editor is not active, an error will occur. An error occurs if <i>scrn_num</i> does not exist.</p>
Examples:	<pre> IF SCRN_GET(PDV_TP) <> SCRN_SYS THEN SCRN_SET(PDV_TP, SCRN_SYS) -- system screen on TP ENDIF IF SCRN_GET(PDV_CRT) <> appl_screen THEN SCRN_SET(PDV_CRT, appl_screen) -- application screen on PC ENDIF </pre>

11.119 SENSOR_GET_DATA Built-In Procedure

The **SENSOR_GET_DATA** will obtain sensor data from integrated sensors.

Calling Sequence:	<code>SENSOR_GET_DATA (sens_read, <,flag> <,arm>)</code>
Parameters:	<p><i>sens_read</i> : ARRAY [6] of REAL [OUT] <i>flag</i> : INTEGER [OUT] <i>arm</i> : INTEGER [OUT]</p>
Comments:	<p><i>sens_read</i> receives the six elements coming from the sensor. The first three represent the translation in the X, Y and Z direction. The last three elements represent the rotations about the X, Y and Z axes.</p>
	<p><i>flag</i> is assigned the value 1 if the data have never been previously read by a SENSOR_GET_DATA statement, 0 otherwise. This parameter is optional. <i>arm</i> is optional and, if present, indicates an arm other than the default arm on which getting the sensor data.</p>
Examples:	<pre> \$SENSOR_ENBL := TRUE ... SENSOR_GET_DATA(sens_read, flag) -- sensor data provides corrections in X,Y and Z directions IF flag=1 THEN WRITE('New data read: ') WRITE('[, sens_read[1], ', sens_read[2], ') WRITE(sens_read[3], ', NL) ENDIF </pre>

11.120 SENSOR_GET_OFST Built-In Procedure

The **SENSOR_GET_OFST** built-in procedure will read the total offsets array representing the real position of the robot with respect to the programmed position.

Calling Sequence:	<code>SENSOR_GET_OFST (ofst_tot <,arm>)</code>
--------------------------	--

Parameters: `ofst_tot : ARRAY [6] of REAL [OUT]`
 `arm : INTEGER [IN]`

Comments: `ofst_tot` receives the six elements coming from the sensor. The first three represent the offsets in the X, Y, and Z directions. The last three elements represent the rotations about the X, Y, and Z axes.
`arm` is optional and if present indicates an arm other than the default arm on which to get the sensor data.

Examples:

```
$SENSOR_ENBL := TRUE
...
SENSOR_GET_OFST(ofst_tot)
WRITE('Actual Offsets: ',NL)
--Case of a sensor providing offsets and rotations
WRITE(['[,ofst_tot[1],',',ofst_tot[2],',',ofst_tot[3],','],NL)
WRITE(['[,ofst_tot[4],',',ofst_tot[5],',',ofst_tot[6],','],NL)
ENDIF
```

11.121 SENSOR_SET_DATA Built-In Procedure

The SENSOR_SET_DATA built-in procedure is used to send offsets to the motion environment.

Calling Sequence: `SENSOR_SET_DATA (err_track <,arm>)`

Parameters: `err_track : ARRAY [6] of REAL [IN]`
 `arm : INTEGER [IN]`

Comments: `err_track` contains the offsets to be applied to the trajectory. The first three represent the translation in the X, Y, and Z directions. The last three elements represent the rotations about the X, Y, and Z axes. Any `err_track` not used elements must be initialized to zero.
`arm` is optional and if present indicates an arm other than the default arm on which to set sensor data.

Examples:

```
err_track[1] := 0; err_track[2] := 0; err_track[3] := 0
err_track[4] := 0; err_track[5] := 0; err_track[6] := 0
$SENSOR_ENBL := TRUE
$SENSOR_TIME := 500 --time between sensor scanning
WHILE (bool)DO
  ...
  --routine that reads two offsets returned by the sensor
  get_corr(err_track[2], err_track[3])
  SENSOR_SET_DATA(err_track)
  DELAY 500 --time between each sensor scan
  ...
ENDWHILE
```

11.122 SENSOR_SET_OFST Built-In Procedure

This built-in routine is useful for the sensor tracking feature when it is necessary to assign initial offsets with respect to the theoretical position of the robot. This built-in can only be executed when there are no running motions.

Calling Sequence: `SENSOR_SET_OFST (pos<,arm_num>)`

Parameters: `pos : POSITION [IN]`
 `arm_num : INTEGER [IN]`

Comments:	pos indicates the position that is used for determining the offsets from the current robot Cartesian position. These offsets are calculated subtracting the current robot Cartesian position and the value specified in the pos parameter. arm_num indicates the arm number. If not specified, the \$PROG_ARM is used.
Examples:	\$SENSOR_SET_OFST (pos_var, 2)

11.123 SENSOR_TRK Built-In Procedure

The SENSOR_TRK built-in procedure, executed only from inside a HOLDABLE program, will select a particular state of sensor tracking. This will allow the specified (or default if unspecified) arm to be under sensor control without programmed movement.

Calling Sequence:	SENSOR_TRK (bool <,arm_num>)
Parameters:	bool : BOOLEAN [IN] arm_num : INTEGER [IN]
Comments:	<p><i>bool</i> is either TRUE or FALSE and indicates which state should be enabled for the sensor tracking.</p> <p><i>arm_num</i> is optional and if present indicates an arm other than the default arm on which to get the sensor data.</p> <p>The effective enabling of this type of tracking depends on the value of \$SENSOR_ENBL. Note that it is not necessary to re-execute SENSOR_TRK(TRUE) after having set \$SENSOR_ENBL to FALSE and then TRUE again.</p> <p>This tracking can be disabled only by the same program that previously enabled it. SENSOR_TRK cannot be executed if this feature has already been enabled by a second HOLDABLE program that is active at the same time.</p>

SENSOR_TRK mode is automatically disabled when the program that enabled it is deactivated.

Examples:	<pre> ROUTINE send_corr BEGIN \$TIMER[1] := 0 --routine that reads the 2 offsets returned by the sensor get_corr(err_track[2], err_track[3]) SENSOR_SET_DATA(err_track) ENABLE CONDITION[1] END send_corr BEGIN CONDITION[1] : WHEN \$TIMER[1] > 500 DO N send_corr --Send offset every 500 ms ENDCONDITION </pre>
------------------	---

```

$SENSOR_TIME := 500 --time between each sensor scanning
MOVE TO p_start
-- Select the sensor tracking mode to put the arm under
-- sensor control without programmed movement
SENSOR_TRK(TRUE)
--Enable sensor tracking
$SENSOR_ENBL := TRUE
ENABLE CONDITION[1]
--This semaphore will be signalled by another program
--or condition.
WAIT sem
$SENSOR_ENBL := FALSE
MOVE TO p_end
.....
  
```

11.124 SIN Built-In Function

The SIN built-in function returns the sine of a specified angle.

Calling Sequence: SIN (angle)

Return Type: REAL

Parameters: angle : REAL [IN]

Comments: angle is measured in degrees.

The returned value will always be in the range of -1.0 to 1.0.

Examples: value := radius * SIN(angle)

```
x := SIN(angle1) * COS(angle2)
```

```
x := SIN(60) -- x = 0.866025
```

11.125 SQRT Built-In Function

The SQRT built-in function returns the square root of a specified number.

Calling Sequence: SQRT (num)

Return Type: REAL

Parameters: num : REAL [IN]

Comments: If num is less than zero, an error will occur.

Examples: legal := SQRT(answer) > 100.0 -- obtains a boolean value

```
x := SQRT(276.971) -- x = 16.6424
```

11.126 STANDBY Built-In Procedure

The STANDBY built-in procedure is useful in case the energy saving function is enabled (\$TUNE[27] > 0), in order to put the arm in the stand-by status when the following conditions are met: the system is in AUTO, the drives are ON and there are no running motions.

Calling Sequence: STANDBY (flag <, arm_num>)

Parameters: flag : BOOLEAN [IN]
arm_num : INTEGER [IN]

Comments: If *flag* is set to ON, the arm immediately enters stand-by state and remains in that state until the next motion is executed or the next calling to the STANDBY procedure with *flag* set to OFF.

If *flag* is set to OFF, the stand-by status of the arm is exited.

arm_num, if present, represents the arm number. If not specified, \$PROG_ARM is used.

Examples: STANDBY(OFF,2) -- request to exit from stand-by status for ARM 2
STANDBY (ON) -- request to enter stand-by status for \$PROG_ARM

11.127 STR_CAT Built-In Function

The STR_CAT built-in function joins two strings. This built-in function applies both to ASCII strings and UNICODE strings.

Calling Sequence: destination_string := STR_CAT (source_string1, source_string2)

Return Type: STRING

Parameters: source_string1 : STRING [IN]
source_string2 : STRING [IN]
destination_string : STRING [OUT]

Comments: *source_string1* is the string which *source_string2* is to be appended to
source_string2 is the string to be appended to *source_string1*

The return value is the result of appending the value of *source_string2* to the value of *source_string1*.

If the resulting string is too long to fit into the *destination_string* variable, the rest of the string is truncated (just like any other string assignment).

Note that UNICODE strings are double + 2 bytes, compared with ASCII strings.

Examples:

```
PROGRAM concat
VAR str : STRING[8]
    s1, s2, s3 : STRING[5]
BEGIN
    s1 := 'Hi '
    s2 := 'Hello '
    s3 := 'There'
    str := STR_CAT(s1, s3)      -- str now equals Hi There
    str := STR_CAT(s2, s3)      -- str now equals Hello Th
END concat
```

11.128 STR_CODING Built-In Function

The STR_CODING built-in function returns an integer value indicating whether the passed string is ASCII or UNICODE.

Calling Sequence: int_coding := STR_CODING (source_string)

Return Type: INTEGER

Parameters: source_string : STRING [IN]
int_coding : INTEGER [OUT]

Comments: *source_string* is the string to be processed by the built-in.
int_coding is the current coding of the *source_string*:
 – 0 - ASCII string
 – 1 - UNICODE string

11.129 STR_CONVERT Built-In Function

The STR_CONVERT built-in function converts the passed string either to UNICODE or to ASCII, depending on the value of the flag.

Calling Sequence: `destination_string := STR_CONVERT (source_string, flag)`

Return Type: STRING

Parameters: *source_string* : STRING [IN]
flag : INTEGER [IN]
destination_string : STRING [OUT]

Comments: *source_string* is the string to be converted
flag indicates which conversion is to be performed:
 – 0 - convert to ASCII
 – 1 - convert to UNICODE
 – -1 - convert to the opposite coding (to UNICODE if ASCII, to ASCII if UNICODE)
 The return value (*destination_string*) is the converted string.

11.130 STR_DEL Built-In Procedure

The STR_DEL built-in procedure deletes a sequence of characters from a STRING.

Calling Sequence: `STR_DEL (source_string, start_index, chars_number)`

Parameters: *source_string* : STRING [IN, OUT]
start_index : INTEGER [IN]
chars_number : INTEGER [IN]

Comments: *source_string* is the string in which one or more characters are to be deleted. If it is uninitialized, an error occurs.

start_index is the index indicating where in *source_string* the deletion will start. If it is greater than the maximum size of *source_string*, the built-in has no effect. If it is less than one, an error occurs.

chars_number is the total number of characters to be deleted. If it is greater than the number of characters from *start_index* to the end of *source_string*, then all of the characters past *start_index* are deleted. If it is less than one, an error occurs.

Examples:

```

PROGRAM str
VAR letters : STRING[10]
BEGIN
  letters := 'abcdefghijkl'
  STR_DEL(letters, 0, 3) -- ERROR: start_idx < 1
  STR_DEL(letters, 1, -1) -- ERROR: length<1
  STR_DEL(letters, 30, 5) -- Nothing happens: 30>max length
  STR_DEL(letters, 4, 2)    -- letters = 'abcfghij'
  letters := 'abcdefghijkl' -- restore for next example
  STR_DEL(letters, 8, 5) -- letters = 'abcdefg'
END str

```

11.131 STR_EDIT Built-In Procedure

The STR_EDIT built-in procedure performs various editing and conversions on the specified string.

Calling Sequence: `STR_EDIT(source_string, operator)`

Parameters: `source_string : STRING [IN, OUT]`
`operator : INTEGER [IN]`

Comments: `source_string` is the STRING to be modified.
`operator` is an INTEGER value which indicates the operation to be performed on the `source_string`. Several string operators can be combined using the OR operator. The following predefined constants represent the different operations to be performed on the STRING:

- `STR_LWR` Converts all upper case characters to lower case.
- `STR_UPR` Converts all lower case characters to upper case.
- `STR_TRIM` Removes leading and trailing blanks, tabs and new line characters.
- `STR_COMP` Converts multiple whitespace characters to a single character.
- `STR_COLL` Removes all whitespace characters from the source string.

Examples: `STR_EDIT(source_string, STR_UPR)`
`STR_EDIT(source_string, STR_TRIM)`
`STR_EDIT(source_string, STR_LWR OR STR_COLL)`

11.132 STR_GET_INT Built-In Function

This routine returns an INTEGER value reading it in a STRING.

Calling Sequence: `integer_val := STR_GET_INT
(source_string<,start_index,bytes_number>)`

Return Type: INTEGER

Parameters: `source_string: STRING [IN]`
`start_index: INTEGER [IN]`
`bytes_number: INTEGER [IN]`
`integer_val : INTEGER [OUT]`

Comments: `source_string` is the STRING from where the INTEGER value is extracted.
`start_index` if not specified, the INTEGER is read from the first byte.
`bytes_number` is the total amount of bytes to be read starting from `start_index`; if not specified, 4 bytes are read. Significant values for this parameter are between 1 and 4.

Examples: `int_val := STR_GET_INT('pippo',2,1) -- read the 2nd byte in
-- pippo string. int_val is set to 105.`

11.133 STR_GET_REAL Built-In Function

This routine returns a REAL value reading it in a STRING.

Calling Sequence: `rea_val := STR_GET_REAL
(source_string<,start_index,bytes_number>)`

Return Type: REAL

Parameters: source_string: STRING [IN]
 start_index: INTEGER [IN]
 bytes_number: INTEGER [IN]
 real_val : REAL [OUT]

Comments: *source_string* is the STRING from where the REAL value is extracted.
start_index if not specified, the REAL is read from the first byte.
bytes_number is the total amount of bytes to be read starting from *start_index*; if not specified, 4 bytes are read. Significant values for this parameter are between 1 and 4.

Examples: *rea_val* := STR_GET_REAL('pippo', 2, 1) -- read the 2nd byte in
-- pippo string.

11.134 STR_INS Built-In Procedure

The STR_INS built-in procedure inserts a sequence of characters into a STRING.

Calling Sequence: STR_INS (source_string, start_index, insert_string)

Parameters: source_string : STRING [IN, OUT]
 start_index : INTEGER [IN]
 insert_string : STRING [IN]

Comments: *source_string* is the STRING to be modified. If it is uninitialized, then *start_index* must be zero or an error occurs.
start_index is an index indicating where, in *source_string*, the new sequence of characters is to be inserted. *start_index* must be between one and the current length of the string. A value out of this range will cause an error even if the string has a maximum length greater than the length of its current value.
insert_string is the new sequence of characters. If the result is greater than the declared length of *source_string*, then it is truncated. *insert_string* is inserted into *source_string*, not written over it.

Examples:

```
PROGRAM instr
VAR str : STRING[20]
BEGIN
  STR_INS(str, 3, 'Specify number') -- ERROR: start_idx > 0
  STR_INS(str, 1, 'Specify number') -- str='Specify number'
  STR_INS(str, 15, 'arm ') -- str='Specify numberarm'
  STR_INS(str, -2, 'arm ') -- ERROR: start is out of range
  STR_INS(str, 9, 'arm ') -- str='Specify arm numberarm'
END instr
```

11.135 STR_LEN Built-In Function

The STR_LEN built-in function returns the current length of a STRING. It applies both to the ASCII strings and to the UNICODE strings.

Calling Sequence: integer_val := STR_LEN (source_string)

Return Type: INTEGER

Parameters: source_string : STRING [IN]
 integer_val : INTEGER [OUT]

Comments: *source_string* is the source string expression. The returned value is not the **declared** length of *source_string*: it is the length of the **current** value.

Examples:

```

PROGRAM strlen
VAR
    str : STRING[10]
    length : INTEGER
BEGIN
    str := 'abcdef' -- initialize str
    length := STR_LEN(str) -- length now equals 6
    str := 'ab' -- change str value
    length := STR_LEN(str) -- length now equals 2
END strlen

```

11.136 STR_LOC Built-In Function

The STR_LOC built-in function returns the location, in a STRING, where a specified sequence of characters begins.

Calling Sequence:

```
integer_val := STR_LOC (original_string, find_me
<,start_from>)
```

Return Type:

INTEGER

Parameters:

original_string : STRING	[IN]
find_me : STRING	[IN]
start_from : INTEGER	[IN]
found_at : INTEGER	[OUT]

Comments:

original_string is the string in which *find_me* is to be searched for.

If *original_string* is empty (s:=’), STR_LOC returns trappable error 39957. An example follows about handling such a situation:

```
.....
VAR ls_device_info : STRING[32]
...
ls_device_info := ''
...
IF (STR_LEN(ls_device_info) > 0) THEN
    IF STR_LOC(ls_device_info, 'UD') <> 0 THEN
        ls_device_info := 'UD:\SYS\CNFG'
    ENDIF
ENDIF
...

```

find_me is the string to be searched for; if *find_me* is not found, zero is returned. *start_from* is an optional integer parameter which indicates the direction of searching and the index where to start searching.

If not specified or 0, *find_me* is searched for starting from the beginning of *original_string*.

If >0, *find_me* is searched, starting from the character which is in the position indicated by *start_from*.

If <0, *find_me* is calculated in the following way: start from the string character whose position is indicated by *start_from* (without sign), from *original_string* beginning. Look for *find_me* substring, passed as parameter (reading it from left to right anyway), going leftwards.

found_at is the position in which *find_me* is found inside the *original_string*.

Examples:

```

PROGRAM tstr_loc NOHOLD
VAR vs_SNr : STRING[64] NOSAVE

```

```

ROUTINE ru_GetSerial(as_str : STRING)
VAR li_idx : INTEGER

```

```

BEGIN
  -- Check if the string is not an empty string
  IF STR_LEN(as_str) > 0 THEN
    -- Look for the first underscore
    li_idx := STR_LOC(as_str, '_')
    IF li_idx > 1 THEN
      -- Remove the header of the string
      STR_DEL(as_str, 1, li_idx)
    ENDIF
  ENDIF
  RETURN
END ru_GetSerial

BEGIN
  vs_SNr := $SYS_ID
  ru_GetSerial(vs_SNr)
  WRITE LUN_CRT ('The serial number of the controller ''',
$SYS_ID, ''' is ', vs_SNr, NL)
END tstr_loc

```

The output result from the listed above program, is:

```
The serial number of the controller 'CNTRLC5G_125' is 125
```

11.137 STR_OVS Built-In Procedure

The STR_OVS built-in procedure replaces a sequence of characters in a STRING with a new sequence of characters.

Calling Sequence: STR_OVS (source_string, start_index, replace_string)

Parameters:	source_string : STRING	[IN, OUT]
	start_index : INTEGER	[IN]
	replace_string : STRING	[IN]

Comments: *source_string* is the STRING to be modified. If this is uninitialized, an error occurs.

start_index is an index indicating where, in *source_string*, the new sequence is to start. *start_index* must be between zero and the current length of the STRING. A value out of this range will cause an error even if the STRING has a maximum length greater than the length of its current value.

replace_string is the new sequence of characters.

If the result is greater than the declared length of *source_string*, then it is truncated.

Examples:

```

PROGRAM over_str
VAR
  source : STRING[10]
BEGIN
  source := 'The Cat'
  STR_OVS(source, 5, 'Dog')  -- source = 'The Dog'
  STR_OVS(source, 1, 'BIG')  -- source = 'BIG Dog'
  STR_OVS(source, 5, 'Chicken')  -- source = 'BIG Chicke'
END over_str

```

11.138 STR_SET_INT Built-In Procedure

This routine sets the "byte" elements of a STRING (at a specified index and for a certain number of bytes) to an INTEGER value. This is not the same as [ENCODE Statement](#) which puts the ASCII representation in the string.

Calling Sequence: STR_SET_INT (source_integer, destination_string<, start_index<, bytes_number>>)

Parameters:

source_integer:	INTEGER	[IN]
destination_string:	STRING	[IN/OUT]
start_index:	INTEGER	[IN]
bytes_number:	INTEGER	[IN]

Comments:

source_integer is the integer value to be copied into the *destination_string* variable *destination_string* is the string in which the integer value is to be placed. *start_index* is the starting index, in *destination_string*, where the value is written. If not specified, the integer value is written starting from the first byte. *bytes_number* is the total amount of bytes in which the *source_integer* value is to be inserted in the *destination_string* (starting from *start_index*); if not specified, 4 bytes are written starting from *start_index*. Valid values for this parameter are included in the range 1..4.

Examples:

```
VAR p: STRING [10]
BEGIN
    p := 'pippo'
    STR_SET_INT (0x61626364,p,3,4) -- p becomes 'pidcba'.
END
```

See also: [STR_SET_REAL Built-In Procedure](#)

11.139 STR_SET_REAL Built-In Procedure

This routine sets the "byte" elements of a STRING (at a specified index and for a certain number of bytes) to a REAL value. This is not the same as [ENCODE Statement](#) which puts the ASCII representation in the string.

Calling Sequence: STR_SET_REAL (source_real, destination_string <, start_index<, bytes_number>>)

Parameters:

source_real:	REAL	[IN]
destination_string:	STRING	[IN/OUT]
start_index:	INTEGER	[IN]
bytes_number:	INTEGER	[IN]

Comments:

source_real is the real value to be copied into the *destination_string* variable. *destination_string* is the string in which the real value is to be placed. *start_index* is the starting index in *destination_string* where the value is written. If not specified, the real value is written starting from the first byte. *bytes_number* is the total amount of bytes in which the *source_real* value is to be inserted in the *destination_string* (starting from *start_index*); if not specified, 4 bytes are written starting from *start_index*. Valid values for this parameter are included in the range 1..4.

Examples:

```
BEGIN
-- Encode a status and a position in a string (very common
-- when handling communication)
```

```

vi_sts := 1
vp_pos := POS(1, 2, 3, 4, 5, 6)
STR_SET_INT(vp_pos, vs, 1, 4)
STR_SET_REAL(vp_pos.X, vs, 5, 4)
STR_SET_REAL(vp_pos.Y, vs, 9, 4)
STR_SET_REAL(vp_pos.Z, vs, 13, 4)
STR_SET_REAL(vp_pos.A, vs, 17, 4)
STR_SET_REAL(vp_pos.E, vs, 21, 4)
STR_SET_REAL(vp_pos.R, vs, 25, 4)
END

```

See also: [STR_SET_INT Built-In Procedure](#)

11.140 STR_TO_IP Built-In Function

This function converts a string containing an IP address notation to an INTEGER value.

Returns the integer representation of the IP address. For example, STR_TO_IP("90.0.0.2") returns 0x5A000002.

This is useful when configuring some of the system parameters that represent IP addresses but take integer values.

If the passed string is not a valid IP address, this function returns the 40109 trappable error.

Calling Sequence: destination_integer := STR_TO_IP(source_string)

Return Type: INTEGER

Parameters: source_string : STRING [IN]
destination_integer : INTEGER [OUT]

Comments:

Examples: int_val:=STR_TO_IP("90.0.0.2") -- int_val is set to
-- 0x200005A value

11.141 STR_TOKENS Built-In Function

The STR_TOKENS built-in function breaks down the supplied source string into separate sub-strings according to the defined delimiter(s), and returns the number of sub-strings found.

Calling Sequence: count := STR_TOKENS(source, delimiter, a_substr<, index>)

Return Type: INTEGER

Parameters: count : INTEGER [OUT]
source : STRING [IN]
delimiter : STRING [IN]
a_substr : ARRAY OF STRING [OUT]
index : INTEGER [IN]

Comments:

source is the STRING to be parsed
delimiter contains the list of characters used to separate each string
a_substr contains the returned result
index: if supplied, it can be used for specifying the start position for parsing
count returns the number of found sub-strings.

Examples:

```

VAR count : INTEGER
    source : STRING[100]
    delimiter : STRING[10]
    a_substr : ARRAY[10] OF STRING[20]

BEGIN
    source := 'abc def ghi'
    delimiter := ' '
    count := STR_TOKENS(source, delimiter, a_substr)
    -- count := 3 and a_substr[1] := 'abc', [2] := 'def' etc

    source := 'abc;def,ghi jkl'
    delimiter := ';' , -- Both ; and , to be delimiters
    -- Starting at index [4]
    count := STR_TOKENS(source, delimiter, a_substr, 4)
    -- count := 3 and a_substr[1] := 'def' [2] := 'ghi jkl'

```

11.142 STR_XTRT Built-In Procedure

The STR_XTRT built-in procedure obtains a substring from a specified STRING.

Calling Sequence: STR_XTRT (source_str, start_index, substring_length, destination_str)

Parameters:

source_string : STRING	[IN]
start_index : INTEGER	[IN]
substring_length : INTEGER	[IN]
destination_string : STRING	[OUT]

Comments: *source_string* is the source STRING which the substring is to be extracted from. It will remain unchanged.

start_index is an index indicating where to start copying. *start_index* must be between zero and the current length of the *source_str*. A value out of this range will cause an error even if the *source_str* has a maximum length greater than the length of its current value.

substring_length is the number of characters to be copied. If it is less than zero, an error will occur. If it is greater than the number of characters from *start_index* to the end of *source_string*, then all of the characters are copied into *destination_string*.

destination_string is the STRING that will hold the copied substring. If the length of the result is greater than the declared length of *destination_string*, then the result is truncated.

Examples:

```

PROGRAM strg
VAR
    str, target : STRING[10]
BEGIN
    str := 'The Cat'
    STR_XTRT(str, 5, 3, target) -- target = 'Cat'
END strg

```

11.143 SYS_ADJUST Built-In Procedure

Reserved.

11.144 SYS_CALL Built-In Procedure

The SYS_CALL built-in procedure performs the specified system command.

Calling Sequence: SYS_CALL (cmnd_str <, param>...)

Parameters: cmnd_str : STRING [IN]
 param : STRING [IN]

Comments: *cmnd_str* is a STRING expression whose value is a list of the single characters that would be required to enter the command from the Teach Pendant or from the PC keyboard (when WinC5G program is active). For example, if the FILER VIEW command is being requested, 'FV' would be used.

Refer to "C5G Control Unit Use" manual for further details.

cmnd_str can include any options that can be specified with the command. Some options are only available when the command is issued from SYS_CALL:

- /4 is useful in viewing commands (like MVP, PV, etc..), for indicating that the data displayed on a window or on a file (\$SYS_CALL_OUT) should stay in 40 characters.
- /T, in DISPLAY commands, allows to direct the output of the SYS_CALL command to the Teach Pendant; by default, DISPLAY commands are directed to the PC video (if WinC5G program is active)./N, in MEMORY LOAD, is used for disabling the saving of the .VAR file when a MEMORY SAVE command will be issued on that program. This option is useful when the variables should only be handled from the .COD program.
- /P in MEMORY LOAD, used for loading in a Permanent way application programs.

param is a STRING expression whose value is a parameter that can be specified with the command.

If *param* contains a directory path specification, a double backslash should be used instead of a single one. For example, file *UD:\usr1\pippo.cod*, when inside a SYS_CALL, should be written as follows:

SYS_CALL ('FD', 'UD:\usr1\pippo.cod')

Multiple *param* parameters, separated by commas, are allowed, as necessary for the command. It is also possible to specify more parameter than those required by the command, but only the significative ones will be considered.

By default, commands issued by SYS_CALL are performed without operator interaction, such as confirmation.

The predefined variable \$SYS_CALL_OUT indicates the LUN on which command output will be written. It can be set only to a system-wide LUN or a LUN opened by the program issuing the SYS_CALL.

The predefined variable \$SYS_CALL_TOUT indicates a timeout for the SYS_CALL. A value of 0 indicates no timeout. If the SYS_CALL built-in does not complete within the specified timeout period, it is canceled.

The predefined variable \$SYS_CALL_STS indicates the status of the last SYS_CALL built-in. There is a \$SYS_CALL_STS for each running program.

The following commands cannot be used with the SYS_CALL built-in:

- PE -- PROGRAM EDIT
- FE -- FILER EDIT
- MT -- MEMORY TEACH
- MD -- MEMORY DEBUG
- UA -- UTILITY APPLICATN
- DCS -- DISPLAY CLOSE SELECT
- FUDC -- FILER UTILITY DIRECTORY CHANGE
- SCK -- SET CNTRLER KEY-LOCK
- SL -- SET LOGIN
- UCP -- UTILITY COMMUNICN PORT_CHAR
- The program does not continue execution until the SYS_CALL built-in completes. However, a SYS_CALL built-in is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the built-in completes.

If an error occurs during SYS_CALL execution, the program is paused. For avoiding the suspension of program execution due to SYS_CALL errors the ERR_TRAP_ON (39960) built-in procedure can be used or bit 6 of \$PROG_CNFQ can be set to 1.

NOTE that it is allowed to use the '!' character as the first character in *cmdn_str* parameter, in order to implicitly enable and disable error trapping. Example:

```
SYS_CALL('!FC', vs_filename, 'LD:\EPL\EPLcfg.xml')
```

means

```
ERR_TRAP_ON (39960)
SYS_CALL('FC', vs_filename, 'LD:\EPL\EPLcfg.xml')
ERR_TRAP_OFF(39960)
```

Examples:

```
SYS_CALL('FC', 'temp', 'data')      -- copies temp to data

OPEN FILE lun_id ('dir.dat', 'RW')
$SYS_CALL_OUT := lun_id      -- directs sys_call output to
dir.dat
SYS_CALL('FV')   -- filer view output to dir.dat
SYS_CALL('ML/V', 'prog3')  -- loads only prog3.var
```

11.145 SYS_SETUP Built-In Procedure

Reserved.

11.146 TABLE_ADD Built-In Procedure

The TABLE_ADD built-in procedure adds a table to the DATA page of the Teach

Pendant. The table must be created by defining a RECORD which fields are the columns of the table.

When TABLE_ADD is called, the new table can be accessed from TP.

Calling Sequence: TABLE_ADD (table_name, owning_program<,file_for_saving>)

Parameters:

table_name: STRING	[IN]
owning_program: STRING	[IN]
file_for_saving: STRING	[IN]

Comments: The parameter *table_name* is the name of the TYPEDEF which defines the table elements.

The TYPEDEF identifies the table, the fields in the TYPEDEF identify the columns of the table.

The rows are represented by the variables declared with such TYPEDEF.

The *owning_program* parameter is the name of the program that owns the table TYPEDEF declaration.

The optional parameter *file_for_saving* can be specified if the table should be saved in a .VAR file different than the owning program.

An error is returned if the owning program does not exist. No error is reported if the .VAR program does not exist as it might need to be created.

Examples:

```
PROGRAM exampetable NOHOLD
TYPE usertable = RECORD
    col1 : INTEGER
    col2 : BOOLEAN
    col3 : POSITION
ENDRECORD
```

```
VAR line1, line2 : usertable
```

```
line3_6 : ARRAY[4] OF usertable
```

```
BEGIN
```

```
    TABLE_ADD('userTable', $PROG_NAME, 'TabUser')
END exampetable
```

See also: [TABLE_DEL Built-In Procedure](#).

[C5G Control Unit Use manual](#), [DATA page](#) par. 16.2 User table creation from DATA environment on page 558.

11.147 TABLE_DEL Built-In Procedure

The TABLE_DEL built-in procedure removes the link of the table with the DATA page of the TP.

It deletes the owning and VAR program for the specified table. If there is no owning or VAR program no error is generated.

Calling Sequence: TABLE_DEL (table_name)

Parameters:

table_name: STRING	[IN]
--------------------	------

Comments: The parameter *table_name* is the name of the TYPEDEF which defines the table elements.

Examples: TABLE_DEL ('userTable')

See also: [TABLE_ADD Built-In Procedure.](#)
C5G Control Unit Use manual, **DATA page** section
[par. 16.2 User table creation from DATA environment on page 558.](#)

11.148 TAN Built-In Function

The TAN built-in function returns the tangent of a specified angle.

Calling Sequence: `TAN (angle)`

Return Type: `REAL`

Parameters: `angle : REAL` [IN]

Comments: `angle` is specified in degrees.

Examples: `x := TAN(1) -- x = 0.01745`
`x := TAN(18.9) -- x = 0.34237`

11.149 TREE_ADD Built-In Procedure

This built-in is for adding children to a node of a tree. It can be used in 2 ways depending on whether the children are to be added to a specific tree or to be left "loose" and in which case the nodes can be added to a tree later.

Calling Sequence: `TREE_ADD(tree_id, <parent_record>, <child_record>, {child_record})`

Parameters: `tree_id: INTEGER` [IN]
`parent_record: ANY CLASS` [IN]
`child_record: ANY CLASS` [IN]

Comments: `tree_id` is the id of the being added tree
`parent_record` is the record of the parent to which the child are to be added
`child_record` is the record of the being added child

Examples: -- Adding some nodes to a specific tree
`TREE_ADD(treeid, ve_root, ve_node1, ve_node2)`

-- Adding some nodes to another node but not to any specific tree yet. In fact the nodes are held by the programs so if the program is deactivated the set of nodes are deleted
`TREE_ADD(0, ve_node1, ve_node1a, ve_node1b)`

-- Later adding these nodes to a tree
`TREE_ADD(treeid, ve_node1)`

See also: [TREE_CLONE Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_DEL Built-In Procedure](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

11.150 TREE_CLONE Built-In Procedure

Copy the contents from one tree to another making a clone of the nodes. The nodes are "added" to the existing tree. To perform a perfect clone use [TREE_DEL Built-In Procedure](#) first.

Calling Sequence: TREE_CLONE (source_treeid, dest_treeid <, source_node <, dest_node>>)

Parameters:

source_treeid: INTEGER	[IN]
dest_treeid: INTEGER	[IN]
source_node: ANY CLASS	[IN]
dest_node: ANY CLASS	[IN]

Comments:

source_treeid is the id of the source tree which will be cloned
dest_treeid is the id of the clone tree
source_node is the source node of the tree from which to copy
dest_node is the node of the destination tree where the copied nodes are to be cloned

See also:

[TREE_ADD Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_DEL Built-In Procedure](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

11.151 TREE_CREATE Built-In Function

The TREE_CREATE built-in procedure creates a tree using the specified information.

Calling Sequence: tree_id := TREE_CREATE(<tree_name<, tree_attrib<, root<, root>>>)

Return Type: INTEGER

Parameters:

tree_name : STRING	[IN]
tree_attrib : INTEGER	[IN]
root : ANY CLASS	[IN]

Comments:

tree_name is the optional name of the being created tree. If not specified then the name of the executing program is used (\$PROG_NAME). If a tree with such a name already exists then a trappable error is reported.

tree_attrib:

- 0x0 : whether program specific - in which case when the program is deactivated the tree is automatically deleted
- 0x1 : global - in which case it has to be explicitly deleted
- 0x2 : local tree and can only be accessed by this program's context
- 0x4 : privilege access in order to access the tree

root is the root of the tree. The roots are optional; if none is specified then [TREE_LOAD Built-In Procedure](#) must be used for creating the root(s). This root cannot be removed. If this root is deleted the tree is deleted. There can be multiple roots to a tree. Note that root nodes cannot be added later - they must be added when the tree is created.

See also:

[TREE_ADD Built-In Procedure](#)
[TREE_CLONE Built-In Procedure](#)
[TREE_DEL Built-In Procedure](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

11.152 TREE_DEL Built-In Procedure

The TREE_DEL built-in procedure deletes nodes from the specified tree. This means that all nodes, leafs, etc., from the specified node down are deleted from the tree. If no node is specified, then it is assumed from the root node and the complete tree is removed.

Calling Sequence: `TREE_DEL (tree_id<, root<, root>)`

Parameters: `tree_id : INTEGER` [IN]
`root : ANY CLASS` [IN]

Comments: `tree_id` is the id of the tree owning the being deleted nodes. If the tree id is 0 it means local "loose" nodes.
`root` is the node on which to start deleting

See also:

[TREE_ADD Built-In Procedure](#)
[TREE_CLONE Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

11.153 TREE_LOAD Built-In Procedure

This built-in can be used for loading in a tree. Note that the tree must have been created before it can be loaded. The root .

Calling Sequence: `TREE_LOAD(tree_id, file_name<, root>)`

Parameters: `tree_id : INTEGER` [IN]
`file_name: STRING` [IN]
`root : ANY CLASS` [IN]

Comments: `tree_id` is the id of the being loaded tree
`file_name` is the name of the file containing the values to be loaded in the tree
`root` is an optional variable specifying on which node the tree is to be loaded

See also:

[TREE_ADD Built-In Procedure](#)
[TREE_CLONE Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_DEL Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

11.154 TREE_NODE_INFO Built-In Procedure

The TREE_NODE_INFO built-in procedure returns information about a node in a tree.

Calling Sequence: `TREE_NODE_INFO(tree_id, node_name, child_count<, parent_name>)`

Parameters:

- `tree_id` : INTEGER
- `node_name`: STRING
- `child_count` : INTEGER
- `parent_name` : STRING

Comments:

- `tree_id` is the id of the tree
- `node_name` is the name of the node
- `child_count` is the number of children
- `parent_name` is the name of the parent node

See also:

- [TREE_ADD Built-In Procedure](#)
- [TREE_CLONE Built-In Procedure](#)
- [TREE_CREATE Built-In Function](#)
- [TREE_DEL Built-In Procedure](#)
- [TREE_LOAD Built-In Procedure](#)
- [TREE_NODE_CHILD Built-In Procedure](#)
- [TREE_REMOVE Built-In Procedure](#)
- [TREE_SAVE Built-In Procedure](#)

11.155 TREE_NODE_CHILD Built-In Procedure

The TREE_NODE_CHILD built-in procedure returns information about a specific child.

Calling Sequence: `TREE_NODE_CHILD(tree_id, node_name, child_idx<, child_name><, data_type<, type_name>>)`

Parameters:

- `tree_id` : INTEGER
- `node_name`: STRING
- `child_idx` : INTEGER
- `child_name`: STRING
- `data_type` : INTEGER
- `type_name` : STRING

Comments:

- `tree_id` is the id of the tree
- `node_name` is the name of the node
- `child_idx` is the index of the child
- `child_name` is the name of the child
- `data_type` is the datatype of the child
- `type_name` is the typename of the child if REC, AREC, NODE, PATH

See also:

- [TREE_ADD Built-In Procedure](#)
- [TREE_CLONE Built-In Procedure](#)
- [TREE_CREATE Built-In Function](#)
- [TREE_DEL Built-In Procedure](#)
- [TREE_LOAD Built-In Procedure](#)
- [TREE_NODE_INFO Built-In Procedure](#)
- [TREE_REMOVE Built-In Procedure](#)
- [TREE_SAVE Built-In Procedure](#)

11.156 TREE_REMOVE Built-In Procedure

The TREE_REMOVE built-in procedure is used for removing nodes from a tree.

Note that all the sub-tree nodes are removed and deleted.

If the tree identifier is not specified, then an error occurs. If an attempt is made to remove the root node, then this is an error as a tree must always have a root.

Calling Sequence: TREE_REMOVE(tree_id, node<, node>)

Parameters: tree_id : INTEGER [IN]
node : ANY CLASS [IN]

Comments: *tree_id* is the id of the being removed tree
node is the being removed node

See also: [TREE_ADD Built-In Procedure](#)
[TREE_CLONE Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_DEL Built-In Procedure](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_SAVE Built-In Procedure](#)

Example:

11.157 TREE_SAVE Built-In Procedure

The TREE_SAVE built-in procedure saves the specified tree structure to a .VTR file.

Calling Sequence: TREE_SAVE(tree_id, file_name<, root>)

Parameters: tree_id : INTEGER [IN]
file_name: STRING [IN]
root : ANY CLASS [IN]

Comments: *tree_id* is the id of the tree;
file_name is the name of the file .VTR where the tree structure is to be saved
root is the tree root

See also: [TREE_ADD Built-In Procedure](#)
[TREE_CLONE Built-In Procedure](#)
[TREE_CREATE Built-In Function](#)
[TREE_DEL Built-In Procedure](#)
[TREE_LOAD Built-In Procedure](#)
[TREE_NODE_INFO Built-In Procedure](#)
[TREE_NODE_CHILD Built-In Procedure](#)
[TREE_REMOVE Built-In Procedure](#).

11.158 TRUNC Built-In Function

The TRUNC built-in function truncates a REAL value to obtain an INTEGER result.

Calling Sequence: TRUNC (value)

Return Type: INTEGER

Parameters: value : REAL [IN]

Comments: value is a REAL expression to be truncated to produce an INTEGER result.
This function can be used to convert a REAL expression into an INTEGER value.

Examples:

```
X := TRUNC(16.35) -- X is assigned 16
ROUTINE real_to_int (value : REAL) : INTEGER
BEGIN
  RETURN(TRUNC(value))
END real_to_int
```

11.159 VAR_CLONE Built-in Procedure

The VAR_CLONE built-in procedure can be used for creating an identical variable, with just a different name and, if wished, a different program.

Calling Sequence: VAR_CLONE (original_prog_name, original_var_name,
new_prog_name, new_var_name, var_attr<, index1<, index2>>)

Parameters:

original_prog_name : STRING	[IN]
original_var_name : STRING	[IN]
new_prog_name : STRING	[IN]
new_var_name : STRING	[IN]
var_attr: INTEGER	[IN]
index1 : INTEGER	[IN]
index2 : INTEGER	[IN]

Comments: original_prog_name is the name of the program owning the original variable
original_var_name is the name of the original variable
new_prog_name is the name of the program owning the new variable (if different than the original owning program)
new_var_name is the name of the new variable
var_attr is the variable attribute specification; can be used for setting attributes of the new variable and optional array indices which can be used for cloning a variable of a different dimension:
0x01 : EXPORTED FROM
0x02 : NOSAVE
0x04 : CONST
0x08 : GLOBAL
0x10 : NODATA
index1 is the first dimension if the variable is an array. In this case, its dimension(s) must be specified; if 0 is specified, it means a dynamic array size; -1 means single.
index2 is the second dimension: if index1 is being specified, then index2 must be specified too. It cannot be -1. If index1 is 0, index2 can be anything as only check if not 0. Note that if the variable is an array, its dimension(s) must be specified.

11.160 VAR_CREATE Built-In Procedure

The VAR_CREATE built-in procedure can be used for creating a variable.

Note that

1. it does not add the declaration clause to the program, it just creates the variable in memory and sets the modified flag

2. if the variable already exists in memory, the user is not warned.

Calling Sequence: VAR_CREATE (prog_name, var_name, var_type, var_attr, index1, index2<, arm_num<, type_name>)

Parameters:

prog_name : STRING	[IN]
var_name : STRING	[IN]
var_type : INTEGER	[IN]
var_attr : INTEGER	[IN]
index1 : INTEGER	[IN]
index2 : INTEGER	[IN]
arm_num : INTEGER	[IN]
type_name : STRING	[IN]

Comments: *prog_name* is the name of the program that owns this variable

var_name is the name of the being created variable

var_type is the type code, as follows:

- 1 - Integer
- 2 - Real
- 3 - Boolean
- 4 - String
- 5 - vector
- 6 - Position
- 7 - Jointpos
- 8 - Xtndpos
- 9 - Record
- 10 - Path
- 11 - Semaphore
- 12 - Node

var_attr is the variable attributes specification:

- 0x01 : EXPORTED FROM
- 0x02 : NOSAVE
- 0x04 : CONST
- 0x08 : GLOBAL
- 0x10 : NODATA

index1 is the first dimension if the variable is an array. If the variable is an array, its dimension(s) must be specified; if 0 is specified, it means a dynamic array size; -1 means single.

index2 is the second dimension: if *index1* is being specified, then *index2* must be specified too. It cannot be -1. If *index1* is 0, *index2* can be anything as only check if not 0. Note that if the variable is an array, its dimension(s) must be specified.

arm_num is the arm number, if the being created variable is a jointpos or xtndpos, or string size; it can be 0 for strings it's optional

type_name is the type name, if the variable type is TYPE; it's optional.

Examples:

```
-- creates the following variable: VAR anInt : INTEGER
VAR_CREATE("pippo", "anInt", 1, 0, -1,-1)

-- creates the following variable:
-- VAR aStr : STRING[10] EXPORTED FROM pippo
VAR_CREATE("pippo", "aStr", 4,1,-1,-1,10)
```

11.161 VAR_INFO Built-In Function

The VAR_INFO built-in function finds out information about a variable, such as its name and its owning program

Calling Sequence: VAR_INFO (anyvar, flags, var_name<, owing_prog>)

Return Type: INTEGER

Parameters:

anyvar : ANY TYPE	[IN]
flags : INTEGER	[IN]
var_name : STRING	[OUT]
owning_prog : STRING	[OUT]

Comments:

any_var is the variable (of any type) about which information can be obtained (such as name and owning program)

flags specifies how the information should be obtained. A value of 0x1 is to obtain the actual local parameter as opposed to finding the program variable being referenced by the local parameter

var_name is the ascii name of the variable

owning_prog is the ascii name of the owning program.

The return value can be:

- 0: no variable found (the variable is a parameter to a routine but the actual variable could not be determined; this can happen, for example, if the parameter to the routine is an expression)
- 1: program variable found
- 2: local variable found
- 3: parameter variable found.

11.162 VAR_LINK Built-In Procedure

The VAR_LINK built-in procedure is the means to link (connect) one variable to another, so that changes in one are also reflected in the other.

Calling Sequence: VAR_LINK (pointer_to, pointer_at, flags, index1, index2)

Parameters:

pointer_to : Variable of any type except Integer, Real and Boolean and cannot be local or argument variable	[IN]
pointer_at : Variable of any type except Integer, Real and Boolean and cannot be local or argument variable	[IN]
flags : INTEGER	[IN]
index1 : INTEGER	[IN]
index2 : INTEGER	[IN]

Comments:

pointer_to is the pointer to the variable to be linked to

pointer_at is the pointer to the variable to be linked at the first one

flags 0x1 Copy value when the link is broken

index1 Array index 1 at which to point at

index2 Array index 2 at which to point at

Links to variable so that changes in one are also reflected in the other

Examples:

```

PROGRAM varlink
TYPE te1 = RECORD
  fi : INTEGER
ENDRECORD
te2 = RECORD
  fi : INTEGER
ENDRECORD
  
```

```

        nd1 = NODEDEF
        $MOVE_TYPE
        fi : INTEGER
        ENDNODEDEF
        VAR i : INTEGER
        r : REAL
        wi, wi1, wi2 : ARRAY[1] OF INTEGER
        xi, xi1, xi2 : ARRAY[2, 3] OF INTEGER
        vs1, vs2 : STRING[40]
        vp1, vp2 : POSITION
        pth : PATH OF nd1
        ne1, ne2 : nd1
        vela, velb : tel
        ve2 : te2
        ROUTINE rr(ap : POSITION)
        BEGIN
            VAR_LINK(vp1, ap)
        END rr

        BEGIN
            -- Error in VAR_LINK
            VAR_LINK(vela, ve2)      -- Different types
            VAR_LINK(wi1, xi1)       -- Different dimensions
            -- Ok
            VAR_LINK(vela, velb)     -- Types
            vela.fi := 12
            WRITE LUN_CRT ('velb=', velb.fi, NL)
            VAR_LINK(wi1, wi2)      -- Array
            wi1[1] := 99
            WRITE LUN_CRT ('wi2[1]=', wi2[1], NL)
            VAR_LINK(vs1, vs2)      -- Strings
            vs1 := 'hello'
            WRITE LUN_CRT ('vs1=', vs2, NL)
            VAR_LINK(ne1, ne2)      -- Node
            ne1.$MOVE_TYPE := LINEAR
            WRITE LUN_CRT ('ne2=', ne1.$MOVE_TYPE, NL)
            rr(vp2)
            vp1 := POS(1, 2, 3, 4, 5, 60, '')
            WRITE LUN_CRT ('VP2:=', vp2, NL)
            VAR_UNLINK(ne1)
            VAR_UNLINK_ALL
        END varlink
    
```

See also:

[VAR_LINK_INFO Built-In Procedure](#), [VAR_LINKS Built-In Procedure](#),
[VAR_LINKSS Built-In Procedure](#), [VAR_UNLINK Built-In Procedure](#),
[VAR_UNLINK_ALL Built-In Procedure](#).

11.163 VAR_LINKS Built-In Procedure

The VAR_LINKS built-in procedure is the means to symbolically link (connect) one variable to another, so that changes in one are also reflected in the other.

Calling Sequence: `VAR_LINKS (pointer_to, pointer_at_name,
pointer_at_program, flags, index1, index2)`

Parameters:

```

pointer_to : Variable of any type except Integer,          [IN]
              Real and Boolean and cannot be local or
              argument variable
pointer_at_name : STRING      [IN]
pointer_at_program : STRING    [IN]
flags : INTEGER      [IN]
index1 : INTEGER      [IN]
index2 : INTEGER      [IN]

```

Comments:

pointer_to is the pointer to the variable to be linked to
pointer_at_name is the name of the variable to be linked at the first one
pointer_at_program is the name of the program which owns the “pointed at” variable
flags 0x1 Copy value when the link is broken
index1 Array index 1 at which to point at
index2 Array index 2 at which to point at

Symbolic links to variable so that changes in one are also reflected in the other.
 This built-in procedure is same as [VAR_LINK Built-In Procedure](#) except the variable being linked to is searched symbolically, providing the program and variable name.

See also:

[VAR_LINK Built-In Procedure](#),
[VAR_LINKSS Built-In Procedure](#),
[VAR_UNLINK Built-In Procedure](#),
[VAR_UNLINK_ALL Built-In Procedure](#).

11.164 VAR_LINKSS Built-In Procedure

The VAR_LINKSS built-in procedure is the means to link (connect) one variable to another, so that changes in one are also reflected in the other. Both variables are symbolically pointed to/at.

Calling Sequence:

```
VAR_LINKSS (pointer_to_name, pointer_to_program,
            pointer_at_name, pointer_at_program, flags, index1, index2)
```

Parameters:

```

pointer_to_name : STRING      [IN]
pointer_to_program : STRING    [IN]
pointer_at_name : STRING      [IN]
pointer_at_program : STRING    [IN]
flags : 0x1 Copy value when the link is broken          [IN]
index1 : Array index 1 at which to point at             [IN]
index2 : Array index 2 at which to point at             [IN]

```

Comments:

pointer_to_name is the name of the variable to be linked to
pointer_to_program is the name of the program which owns the “pointed to” variable
pointer_at_name is the name of the variable to be linked at the first one
pointer_at_program is the name of the program which owns the “pointed at” variable
flags 0x1 Copy value when the link is broken
index1 Array index 1 at which to point at
index2 Array index 2 at which to point at
 Symbolic pointing to and pointing at variable, so that changes in one are also reflected in the other.
 This built-in procedure is same as [VAR_LINKS Built-In Procedure](#) except also the variable being linked to is

searched symbolically, providing the program and variable name.

See also: [VAR_LINK Built-In Procedure](#),
[VAR_LINKS Built-In Procedure](#),
[VAR_UNLINK Built-In Procedure](#),
[VAR_UNLINK_ALL Built-In Procedure](#).

11.165 VAR_LINK_INFO Built-In Procedure

The `VAR_LINK_INFO` built-in Procedure finds out information about a link, such as its name and its owning program

Calling Sequence:	VAR_LINK_INFO (pointer_to, name_of_variable, name_of_program)
Return Type:	INTEGER
Parameters:	<p>pointer_to : Variable of any type except Integer, [IN] Real and Boolean and cannot be local or argument variable</p> <p>name_of_variable : STRING [OUT]</p> <p>name_of_program : STRING [OUT]</p>
Comments:	<p><i>pointer_to</i> is the variable (of any type) about which information can be obtained (such as name and owning program)</p> <p><i>name_of_variable</i> is the ascii name of the variable</p> <p><i>name_of_program</i> is the ascii name of the owning program.</p>
See also:	VAR_LINK Built-In Procedure , VAR_LINKS Built-In Procedure , VAR_LINKSS Built-In Procedure , VAR_UNLINK Built-In Procedure , VAR_UNLINK_ALL Built-In Procedure .

11.166 VAR_UNINIT Built-In Function

The `VAR_UNINIT` built-in function tests a variable reference to see if it is uninitialized and returns a `BOOLEAN` result.

Calling Sequence:	VAR_UNINIT (var_ref)
Return Type:	BOOLEAN
Parameters:	var_ref : any variable reference [IN]
Comments:	<p><i>var_ref</i> is the variable to be tested. It can be a single variable reference (<i>my_var</i>), an array element reference (<i>ary_var[43]</i>), or a field reference (<i>fld_var.fld_name</i>).</p> <p>If <i>var_ref</i> has not been given a value, TRUE is returned. Otherwise, FALSE is returned.</p> <p>If an ARRAY variable is used, it must be subscripted.</p> <p>If a RECORD or NODE variable is used, it must have a field specification.</p> <p>If a PATH variable is specified, the whole path is tested. A PATH is considered uninitialized if it has zero nodes.</p> <p>If a PATH node is tested, a specific field must be specified.</p>
Examples:	<pre>PROGRAM testinit VAR ok : BOOLEAN count : INTEGER (0)</pre>

```

pallet : ARRAY[4,2] OF BOOLEAN
spray_pth : PATH
BEGIN
  -- ok will be set FALSE since count is initialized
  ok := VAR_UNINIT(count)
  -- test an array element
  ok := VAR_UNINIT(pallet[1, 2])
  -- test an entire path
  ok := VAR_UNINIT(spray_pth)
END testinit

```

11.167 VAR UNLINK Built-In Procedure

The VAR_UNLINK built-in procedure breaks up the link between variables.

Calling Sequence: VAR_UNLINK (pointer_to)

Parameters: pointer_to : Variable of any type except Integer, Real and Boolean and cannot be local or argument variable [IN]

Comments: Breakup a link between variables

See also: [VAR_LINK Built-In Procedure](#)

11.168 VAR UNLINK ALL Built-In Procedure

The VAR UNLINK ALL built-in procedure breaks up all links created by this program.

Calling Sequence: VAR_UNLINK_ALL

Parameters: none

Comments: Break up all links created by this program. . The same is also performed when the program is deactivated

See also: [VAR_LINK Built-In Procedure](#), [VAR_UNLINK Built-In Procedure](#).

11.169 VEC Built-In Function

The VEC built-in function returns a VECTOR created from the three specified REAL components.

Calling Sequence: VEC (x, y, z)

Return Type: VECTOR

Parameters: x : REAL [IN]
 y : REAL [IN]
 z : REAL [IN]

Comments: x , y , and z represent the Cartesian components from which the returned VECTOR is composed.

Examples: `v1 := VEC(0, 100, 0)` -- creates vector

11.170 VOL_SPACE Built-In Procedure

The VOL_SPACE built-in procedure checks a specified volume for statistics regarding its usage.

Calling Sequence: VOL_SPACE (device, total, free, volume)

Parameters:

device	: STRING	[IN]
total	: INTEGER	[OUT]
free	: INTEGER	[OUT]
volume	: STRING	[OUT]

Comments: device specifies the name of the device containing the volume to be checked.

The following device names can be used:

UD:, TD:

A volume is the media on a device. For example a particular disk in a disk drive device.

total is set to the total number of bytes used on the volume.

free is set to the total number of unused bytes on the volume.

volume is set to the volume label. An empty STRING indicates the volume is not labeled.

Examples: VOL_SPACE ('UD:', total, free, volume)

11.171 WIN_ATTR Built-In Procedure

The WIN_ATTR built-in procedure sets up video attributes for a specified window.

Calling Sequence: WIN_ATTR (attributes <, win_name>)

Parameters:

attributes	: INTEGER	[IN]
win_name	: STRING	[IN]

Comments: attributes is an INTEGER mask indicating the attributes to be set for the specified window. The following predefined constants represent these attributes:

- WIN_REVERSE -- reverse video
- WIN_BLINK_ON -- turns blinking on
- WIN_BLINK_OFF -- turns blinking off
- WIN_BOLD_ON -- turns bolding on
- WIN_BOLD_OFF -- turns bolding off
- WIN_CRSR_OFF -- turns off display of the cursor
- WIN_CRSR_ON -- turns on display of the cursor

A sequence of attributes can be joined together using the OR operator.

The optional parameter win_name can be used to specify the window for which the attributes are to be set. If it is not specified, the default window indicated by \$DFT_DV[1] is used.

win_name can be one of the following system defined window names or any user-defined window name:

- 'TP:' -- scrolling window on system screen
- 'TP1:' -- window 1 on user screen on Teach Pendant
- 'TP2:' -- window 2 on user screen on Teach Pendant
- 'TP3:' -- window 3 on user screen on Teach Pendant
- 'CRT:' -- scrolling window on system screen of the PC video
-- (when WinC5G program is active)
- 'CRT1:' -- window 1 on user screen on PC video
- 'CRT2:' -- window 2 on user screen on PC video

- ‘CRT3:’ -- window 3 on user screen on PC video

```

Examples: PROGRAM wattr NOHOLD
VAR lun : INTEGER
BEGIN
    OPEN FILE lun ('crt2:', 'rw')      Window 2 on user screen of the PC
    WRITE lun ('This is ')
    WIN_ATTR(WIN_BLINK_ON, 'crt2:')      Turn blink on
    WRITE lun (BLINK, NL)
    WIN_ATTR(WIN_BLINK_OFF, 'crt2:')     Turn blink off
        Reverse video and bold
    WIN_ATTR(WIN_REVERSE OR WIN_BOLD_ON, 'crt2:')
    WRITE lun ('This is REVERSE and BOLD', )
    WIN_ATTR(WIN_REVERSE, 'crt2:')       Reverse video back to normal
    WRITE lun ('this is only BOLD.', NL)
    WIN_ATTR(WIN_BOLD_OFF, 'crt2:')     Turn bold off
    CLOSE FILE lun
END wattr

```

The output from the example program is show below:

This is BLINK
This is REVERSE and BOLD
this is only BOLD

11.172 WIN CLEAR Built-In Procedure

The WIN_CLEAR built-in procedure clears a specified window.

Calling Sequence: WIN CLEAR (clear spec <, win name>)

Parameters: clear_spec : INTEGER [IN]
win_name : STRING [IN]

Comments: *clear_spec* is an INTEGER indicating the portion of the window to be cleared. The following predefined constants can be used:

- WIN_CLR_ALL -- clears entire window
 - WIN_CLR_LINE -- clears line cursor is on
 - WIN_CLR_BOLN -- clears from cursor to beginning of line
 - WIN_CLR_BOW -- clears from cursor to beginning of window
 - WIN_CLR_EOLN -- clears from cursor to end of line
 - WIN_CLR_EOW -- clears from cursor to end of window

The optional parameter *win_name* can be used to specify the window to be cleared. If it is not specified, the default window indicated by \$DFT_DV[1] is used.

win_name can be one of the following system defined window names or any user-defined window name:

- 'TP:' -- scrolling window on system screen
 - 'TP1:' -- window 1 on user screen on Teach Pendant
 - 'TP2:' -- window 2 on user screen on Teach Pendant
 - 'TP3:' -- window 3 on user screen on Teach Pendant
 - 'CRT:' -- scrolling window on system screen of the PC
 - (whenWinC5G program is active)
 - 'CRT1:' -- window 1 on user screen on PC video
 - 'CRT2:' -- window 2 on user screen on PC video

- ‘CRT3:’ -- window 3 on user screen on PC video

Examples: WIN_CLEAR (WIN_CLR_ALL, ‘TP1:’)

11.173 WIN_COLOR Built-In Procedure

The WIN_COLOR built-in procedure sets the foreground and background colors for a specified window.

Calling Sequence: WIN_COLOR (fore_spec, back_spec, all_flag <, win_name>)

Parameters:

fore_spec	: INTEGER	[IN]
back_spec	: INTEGER	[IN]
all_flag	: BOOLEAN	[IN]
win_name	: STRING	[IN]

Comments: *fore_spec* and *back_spec* are INTEGERs indicating the foreground and background colors to be set for the specified window. The following predefined constants can be used:

- WIN_BLACK
- WIN_RED
- WIN_BLUE
- WIN_MAGENTA
- WIN_GREEN
- WIN_YELLOW
- WIN_CYAN
- WIN_WHITE

When only one of the two colours needs to be changed, -1 should be used for the colour that remains unchanged.

If *all_flag* is TRUE, the color change affects all characters on the screen. If it is FALSE, only new characters that appear on the screen are affected.

The optional parameter *win_name* can be used to specify the window for which the colors are to be set. If it is not specified, the default window indicated by \$DFT_DV[1] is used.

win_name can be one of the following system defined window names or any user-defined window name:

- ‘TP:’ -- scrolling window on system screen
- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on system screen of the PC video
 -- (when WinC5G program is active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

Examples:

```

PROGRAM wcolor NOHOLD
VAR i : INTEGER
    lun : INTEGER
BEGIN
    OPEN FILE lun ('CRT2:', 'rw')    -- Window 2 on PC screen
    WRITE lun ('This is ')
    WIN_COLOR(WIN_WHITE, WIN_BLUE, FALSE, 'CRT2:')
    WRITE lun ('WHITE on BLUE', NL)
    WIN_COLOR(WIN_WHITE, WIN_BLACK, FALSE, 'CRT2:')

```

```

  WRITE lun ('This is ')
  WIN_COLOR(WIN_BLACK, WIN_RED, FALSE, 'CRT2:')
  WRITE lun ('BLACK on RED', NL)
  FOR i := 1 TO 3 DO
    WRITE lun ('This is still BLACK on RED', NL)
  ENDFOR
  DELAY 5000 -- then change color of the entire window
  WRITE lun ('Now change entire window to WHITE on BLACK', NL)
  WIN_COLOR(WIN_WHITE, WIN_BLACK, TRUE, 'CRT2:')
  CLOSE FILE lun
END wcolor

```

11.174 WIN_CREATE Built-In Procedure

The WIN_CREATE built-in procedure creates a user-defined window.

Calling Sequence: `WIN_CREATE (win_name, dev_num, attributes, num_rows)`

Parameters:

<code>win_name</code>	: STRING	[IN]
<code>dev_num</code>	: INTEGER	[IN]
<code>attributes</code>	: INTEGER	[IN]
<code>num_rows</code>	: INTEGER	[IN]

Comments: `num_rows` indicates the number of rows the window will occupy. Windows created for the PDV_CRT cannot have more than 25 rows and windows created for the PDV_TP cannot have more than 16 rows.

Created windows are not automatically displayed. [WIN_DISPLAY Built-In Procedure](#) and [WIN_POPUP Built-In Procedure](#) can be used to display created windows.

The programmer is responsible for managing user-defined windows, including cleaning up windows when a program terminates.

A LUN must be opened on a user-defined window before any reads or writes can take place.

`win_name` is a STRING used to identify the window. It follows the naming convention of system defined windows ('xxxx:'). `win_name` can be used in other built-in routines requiring a window name parameter.

`dev_num` indicates the device on which the specified window can be displayed.

The following predefined constants represent devices:

- PDV_TP -- Teach Pendant
- PDV_CRT -- PC screen, when using WinC5G Program
- `attributes` is an INTEGER mask indicating the fixed attributes to be set for the specified window.

The following predefined constants represent these fixed attributes:

- WIN_SCROLL -- output to the window will scroll
- WIN_WRAP -- lines longer than the screen width will wrap to the next line

Examples:

- WIN_WHITE -- foreground color
- WIN_BLACK -- background color
- WIN_BOLD_OFF -- no bolding
- WIN_BLINK_OFF -- no blinking

`WIN_CRSR_ON` -- cursor is displayed-- Create new user defined windows for user screen

`WIN_CREATE('USR1:', PDV_CRT, WIN_SCROLL OR WIN_WRAP, 25)`

`WIN_CREATE('POP1:', PDV_CRT, WIN_SCROLL OR WIN_WRAP, 10)`

-- Set color and attributes for user windows

`WIN_COLOR(WIN_WHITE, WIN_BLUE, TRUE, 'USR1:')`

```

WIN_ATTR(WIN_BOLD_ON, 'USR1:')
WIN_COLOR(WIN_BLACK, WIN_RED, TRUE, 'POP1:')
-- Remove system defined windows from user screen
WIN_REMOVE('CRT1:')
WIN_REMOVE('CRT2:')
WIN_REMOVE('CRT3:')
WIN_DISPLAY('USR1:', SCRN_USER, 0)

```

11.175 WIN_DEL Built-In Procedure

The WIN_DEL built-in procedure deletes the specified user-defined window.

Calling Sequence: WIN_DEL (win_name)

Parameters: win_name : STRING [IN]

Comments: *win_name* can be any user-defined window name (created using the WIN_CREATE built-in routine).

A window can be deleted only after it has been removed from the screen. Deleting a window means the window name can no longer be used in window-related built-in routines.

An error occurs if a window is deleted before all LUNs opened on the window are closed. In addition, it must also be detached.

System defined windows cannot be deleted. *win_name* is not permitted to be deleted if the alphabetical menu window ('TP0:') is currently a popup window on *win_name*.

Examples:

```

WIN_DEL ('menu:')
-- popup window over window USR1
WIN_POPUP('POP1:', 'USR1:')
-- open a lun on window POP1
OPEN FILE lun ('POP1:', 'rw')
FOR i := 1 TO 10 DO
    WRITE lun (i, ': This is an example of a popup window', NL)
ENDFOR
CLOSE FILE lun
-- let user read the message
DELAY 5000
Remove and delete window POP1 from user screen
WIN_REMOVE('POP1:')
WIN_DEL('POP1:')

```

11.176 WIN_DISPLAY Built-In Procedure

The WIN_DISPLAY built-in procedure displays the specified window as a fixed window on the specified screen at the specified row.

Calling Sequence: WIN_DISPLAY (win_name, scrn_num, row_num)

Parameters: win_name : STRING [IN]

 scrn_num : INTEGER [IN]

 row_num : INTEGER [IN]

Comments: *win_name* can be one of the following system defined window names or any user-defined window name:

- 'TP:' -- scrolling window on system screen
- 'TP1:' -- window 1 on user screen on Teach Pendant

- 'TP2:' -- window 2 on user screen on Teach Pendant
 - 'TP3:' -- window 3 on user screen on Teach Pendant
 - CRT: -- scrolling window on system screen of PC video
 - (when WinC5G program is active)
 - CRT1: -- window 1 on user screen on PC video
 - CRT2: -- window 2 on user screen on PC video
 - CRT3: -- window 3 on user screen on PC video
- scrn_num* indicates the screen (user-created or system created) on which the window is to be displayed. System created screens are represented by the following predefined constants:
- SCRN_USER— user screen
 - SCRN_SYS— system screen
- The window is displayed on the device for which it was created, either PDV_TP or PDV_CRT.
- row_num* indicates the number of the row at which the window is to start. If *row_num* causes an overlap of windows or there isn't enough room on the screen for the window, an error occurs.
- A window can be displayed only once on the same screen. It can be displayed on more than one screen at a time.

Examples:

```

WIN_DISPLAY ('menu:', SCRN_USER, 1)

-- Create new user defined windows for user screen
WIN_CREATE('USR1:', PDV_CRT, WIN_SCROLL OR WIN_WRAP, 25)
WIN_CREATE('POP1:', PDV_CRT, WIN_SCROLL OR WIN_WRAP, 10)
-- Set color and attributes for user windows
WIN_COLOR(WIN_WHITE, WIN_BLUE, TRUE, 'USR1:')
WIN_ATTR(WIN_BOLD_ON, 'USR1:')
WIN_COLOR(WIN_BLACK, WIN_RED, TRUE, 'POP1:')
-- Remove system defined windows from user screen
WIN_REMOVE('CRT1:')
WIN_REMOVE('CRT2:')
WIN_REMOVE('CRT3:')
-- display the newly created window
WIN_DISPLAY('USR1:', SCRN_USER, 0)

```

11.177 WIN_GET_CRSR Built-In Procedure

The WIN_GET_CRSR built-in procedure obtains the row and column of the cursor on the specified window.

Calling Sequence: `WIN_GET_CRSR (row, col <, win_name>)`

Parameters:	<code>row</code> : INTEGER	[OUT]
	<code>col</code> : INTEGER	[OUT]
	<code>win_name</code> : STRING	[IN]

Comments: *row* and *col* are assigned the current row and column position of the cursor on the specified window.

The home position (top, left corner) on a window is (0,0). The optional parameter *win_name* can be used to specify the window. If it is not specified, the default window indicated by \$DFT_DV[1] is used.

win_name can be one of the following system defined window names or any user-defined window name:

- 'TP:' -- scrolling window on system screen

- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on system screen of the PC video (when WinC5G program is active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

Examples:

```

OPEN FILE crt1_lun ('CRT1:', 'RW')
row := 0
col := 5
WIN_SET_CRSR (row, col, 'CRT1:') -- sets to 0,5
WHILE row <= max_row DO
    WRITE (msg_str)
    row := row + 1
    col := 5

    WIN_SET_CRSR (row, col, 'CRT1:')
    WIN_GET_CRSR (row, col, 'CRT1:')
ENDWHILE

```

11.178 WIN_LINE Built-In Function

The WIN_LINE built-in function returns the sequence of characters currently displayed at a given location on a specified window.

Calling Sequence:	WIN_LINE <(row <, column <, num_chars <, win_name)>>>												
Return Type:	STRING												
Parameters:	<table border="0"> <tr> <td>row</td><td>: INTEGER</td><td>[IN]</td></tr> <tr> <td>col</td><td>: INTEGER</td><td>[IN]</td></tr> <tr> <td>num_chars</td><td>: INTEGER</td><td>[IN]</td></tr> <tr> <td>win_name</td><td>: STRING</td><td>[IN]</td></tr> </table>	row	: INTEGER	[IN]	col	: INTEGER	[IN]	num_chars	: INTEGER	[IN]	win_name	: STRING	[IN]
row	: INTEGER	[IN]											
col	: INTEGER	[IN]											
num_chars	: INTEGER	[IN]											
win_name	: STRING	[IN]											
Comments:	<p><i>row</i> is the row position inside <i>win_name</i>. If <i>row</i> is not specified or it has a negative value, the current row will be used.</p> <p><i>col</i> is the column position inside <i>win_name</i>. If <i>col</i> is not specified or it has a negative value, the current column will be used.</p> <p><i>num_chars</i> indicates the number of characters to be obtained.</p> <p>If <i>num_chars</i> is not specified or it has a negative value, the entire row is obtained.</p> <p><i>win_name</i> is the name of the window from which the characters are to be obtained. If not specified, the default window is used.</p>												
Examples:	<pre> PROGRAM winline NOHOLD VAR gs_line : ARRAY[20] OF STRING[80] vi_lun : INTEGER BEGIN OPEN FILE vi_lun ('CRT:', 'w') . . -- get row 4 gs_line := WIN_LINE(4, 0, -1, 'CRT:') . . END winline </pre>												

11.179 WIN_LOAD Built-In Procedure

The WIN_LOAD built-in procedure loads the contents of a saved window file into the specified window.

When using the WIN_LOAD built-in routine to load a window from a file onto a PC screen (when WinC5G program is active) the file must have been saved (WIN_SAVE) from a PC screen too.

Calling Sequence: `WIN_LOAD (file_name, win_name <, start_row <, start_col>>)`

Parameters:

<code>file_name</code> : STRING	[IN]
<code>win_name</code> : STRING	[IN]
<code>start_row</code> : INTEGER	[IN]
<code>start_col</code> : INTEGER	[IN]

Comments:

`file_name` is the name of a saved window file that has been created by the WIN_SAVE built-in. The entire contents of this file will be loaded upon the window indicated by `win_name`.

`win_name` is the name of the window upon which the contents of the saved window file will be loaded.

`start_row` is the starting row position inside `win_name`. If `start_row` is not specified, row 0 will be used. If `start_row` is -1, the starting row position is obtained from `file_name` which means the contents will be loaded into the same rows from which they were saved.

If `start_row` is less than -1 or greater than the number of rows on `win_name`-1 (the first row is 0), an error occurs.

The entire contents of `file_name` are loaded, the number of rows to be loaded is obtained from the file. However, if this is greater than the number of rows on `win_name` - `start_row`, an error occurs.

`start_col` is the starting column position inside `win_name`. If `start_col` is not specified, column 0 will be used. If `start_col` is -1, the starting column position is obtained from `file_name` which means the contents will be loaded into the same columns from which they were saved.

If `start_col` is less than -1 or greater than the number of columns on `win_name`-1 (the first column is 0) an error occurs.

Since the entire contents of `file_name` are loaded, the number of columns to be loaded is obtained from the file. However, if this is greater than the number of columns on `win_name` - `start_col`, an error occurs.

Examples:

```
PROGRAM winsl NOHOLD
BEGIN
    -- Saves 5 rows and 50 cols, starting at row 1 and col 5
    WIN_SAVE('win1.win', 'CRT:', 1, 5, 5, 50)
    WIN_CLEAR(WIN_CLR_ALL, 'CRT2:')
    -- Display the saved window on CRT2:, at row 1 and col 5
    WIN_LOAD('win1.win', 'CRT2:', -1, -1)
    WIN_CLEAR(WIN_CLR_ALL, 'CRT2:')
    -- Display the saved window on CRT2:, at row 0 and col 5
    WIN_LOAD('win1.win', 'CRT2:', 0, -1)
    WIN_CLEAR(WIN_CLR_ALL, 'CRT2')
    -- Display the saved window on CRT2:, at row 10 and col 0
    WIN_LOAD('win1.win', 'CRT2:', 10, 0)
    WIN_CLEAR(WIN_CLR_ALL, 'CRT2')
    -- Display the saved window on CRT2:, at row 0 and col 0
    WIN_LOAD('win1.win', 'CRT2:')
END winsl
```

11.180 WIN_POPUP Built-In Procedure

The WIN_POPUP built-in procedure displays the specified window as a popup window on top of the specified fixed window.

Calling Sequence: WIN_POPUP (pop_win_name, fix_win_name <, scrn_num>)

Parameters:

pop_win_name : STRING	[IN]
fix_win_name : STRING	[IN]
scrn_num : INTEGER	[IN]

Comments:

pop_win_name is popped (overlaid) on top of *fix_win_name*.
pop_win_name can be one of the following system defined window names or any user-defined window name:

- ‘TP:’ -- scrolling window on system screen
- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on system screen of PC video (when WinC5G program is active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

fix_win_name can be the name of any window that has been displayed as a fixed window.

If more than one window is popped on top of a single fixed window, the popup windows are tiled vertically. For example, the first one starts at row 0 of *fix_win_name* and the next one starts with the first available row after the first popup window.

An error occurs if *pop_win_name* won’t fit on the remaining space of *fix_win_name*.

The optional parameter *scrn_num* can be used to indicate the screen on which the window is to be popped up. *scrn_num* can indicate a system created screen or a user-created screen. System created screens are represented by the following predefined constants:

- SCRNUSE user screen
- SCRNSYS system screen

A *scrn_num* parameter is only needed if *fix_win_name* is displayed on more than one screen.

Examples:

```

WIN_POPUP ('emsg:', 'menu:')
-- popup window over window USR1
WIN_POPUP('POP1:', 'USR1:')
-- open a lun on window POP1
OPEN FILE lun ('POP1:', 'rw')
FOR i := 1 TO 10 DO
    WRITE lun (i, `: This is an example of a popup window`, NL)
ENDFOR
CLOSE FILE lun
-- let user read the message
DELAY 5000
-- Remove and delete window POP1 from user screen
WIN_REMOVE('POP1:')
WIN_DEL('POP1:')

```

11.181 WIN_REMOVE Built-In Procedure

The WIN_REMOVE built-in procedure removes the specified window from the screen.

WIN_REMOVE Built-In Procedure

Calling Sequence: `WIN_REMOVE (win_name <, scrn_num>)`

Parameters: `win_name : STRING` [IN]
`scrn_num : INTEGER` [IN]

Comments: `win_name` can be one of the following system defined window names or any user-defined window name representing a window that is currently displayed or popped up:

- 'TP:' -- scrolling window on system screen
- 'TP1:' -- window 1 on user screen on Teach Pendant
- 'TP2:' -- window 2 on user screen on Teach Pendant
- 'TP3:' -- window 3 on user screen on Teach Pendant
- 'CRT:' -- scrolling window on system screen of PC video (when WinC5G program is active)
- 'CRT1:' -- window 1 on user screen on PC video
- 'CRT2:' -- window 2 on user screen on PC video
- 'CRT3:' -- window 3 on user screen on PC video

Removing a fixed window causes the portion of the screen occupied by that window to be set to black. Any windows that have been popped up on that window are also removed.

Removing a popped up window causes the underlying fixed window to become visible.

The optional parameter `scrn_num` can be used to indicate the screen from which the window is to be removed. Either a system created or user-created screen can be specified. System created screens are represented by the following predefined constants:

- SCRN_USER -- user screen
- SCRN_SYS -- system screen
- A `scrn_num` parameter is only needed if `win_name` is displayed on more than one screen.

Examples:

```

WIN_REMOVE ('menu:')
-- popup window over window USR1
WIN_POPUP('POP1:', 'USR1:')
-- open a lun on window POP1
OPEN FILE lun ('POP1:', 'rw')
FOR i := 1 TO 10 DO
  WRITE lun (i, ': This is an example of a popup window', NL)
ENDFOR
CLOSE FILE lun
-- let user read the message
DELAY 5000
-- Remove and delete window POP1 from user screen
WIN_REMOVE('POP1:')
WIN_DEL('POP1:')

```

11.182 WIN_SAVE Built-In Procedure

The WIN_SAVE built-in procedure saves all or part of a window to a saved window file. When using WIN_SAVE to save the contents of a window opened onto a PC screen (when WinC5G program is active), the created file will only be compatible with C5G

systems that use WinC5G program Terminal opened.

Calling Sequence: WIN_SAVE (file_name, win_name <, start_row <, start_col <, num_rows <, num_cols <, output_row <, output_col>>>>)

Parameters:

file_name	:	STRING	[IN]
win_name	:	STRING	[IN]
start_row	:	INTEGER	[IN]
start_col	:	INTEGER	[IN]
num_rows	:	INTEGER	[IN]
num_cols	:	INTEGER	[IN]
output_row	:	INTEGER	[IN]
output_col	:	INTEGER	[IN]

Comments: *file_name* is the name of the saved window file that will be used to save the contents of the window.

win_name is the name of the window to be saved to the saved window file.

start_row is the starting row position inside the window. If this parameter is not specified, the first row of the window will be used (row 0). An error occurs if *start_row* is less than 0 or greater than the last row of *win_name*.

start_col is the starting column position inside the window. If this parameter is not specified, column 0 will be used. An error occurs if *start_col* is less than 0 or greater than the last column of *win_name*.

num_rows is the number of rows from the window to save in the saved window file. If this parameter is not specified, all rows from the window will be saved. An error occurs if *num_rows* is less than 1 or greater than the number of rows on *win_name* - *start_row*.

num_cols is the number of columns from the window to save in the saved window file. If this parameter is not specified, all columns from the window will be saved. An error occurs if *num_cols* is less than 1 or greater than the number of columns on *win_name* - *start_col*.

output_row is the starting row on the output screen when the output device is a window.

output_col is the starting column on the output screen when the output device is a window.

Examples:

```

PROGRAM wins1 NOHOLD
CONST ks_dev = 'CRT2:' -- Window for the demo.
VAR vi_i, vi_lun : INTEGER
BEGIN
-- Start off with a clean window
    WIN_CLEAR(WIN_CLR_ALL, ks_dev)
    WIN_COLOR(WIN_RED, WIN_BLACK, FALSE, ks_dev)
    OPEN FILE vi_lun ('CRT2:', 'w')
-- Start in the corner to draw a box
    WIN_SET_CRSR(0, 0, ks_dev)
    WRITE vi_lun ('\201')
    FOR vi_i := 1 TO 10 DO
        WRITE vi_lun ('\205')
    ENDFOR
    WRITE vi_lun ('\187')
-- Middle lines of box
    FOR vi_i := 1 TO 5 DO
        WIN_COLOR(WIN_RED, WIN_BLACK, FALSE, ks_dev)
        WIN_SET_CRSR(vi_i, 0, ks_dev)
        WRITE vi_lun ('\186')
        WIN_COLOR(1, WIN_GREEN, FALSE, ks_dev)
        WRITE vi_lun ('`:::10')
    
```

```

    WIN_COLOR(WIN_RED, WIN_BLACK, FALSE, ks_dev)
    WRITE vi_lun ('\\186')
  ENDFOR
  -- Bottom line of box
  WIN_SET_CRSR(5, 0, ks_dev)
  WRITE vi_lun ('\\200')
  FOR vi_i := 1 TO 10 DO
    WRITE vi_lun ('\\205')
  ENDFOR
  WRITE vi_lun ('\\188')
  -- Save the pattern into two files
  WIN_SAVE('winex.win', ks_dev, 0, 0, 6, 12)
  CYCLE
  WIN_LOAD('winex.win', ks_dev, $CYCLE MOD 15, $CYCLE MOD
69)
  DELAY 150
  IF $CYCLE MOD 69 = 0 THEN
    WIN_COLOR($CYCLE MOD 8, $CYCLE MOD 8 + 1, TRUE, ks_dev)
  ENDIF
END wins1

```

11.183 WIN_SEL Built-In Procedure

The WIN_SEL built-in procedure selects the specified user-defined window for input.

Calling Sequence: WIN_SEL (*win_name* <, *scrn_num*>)

Parameters: *win_name* : STRING [IN]
scrn_num : INTEGER [IN]

Comments: *win_name* can be any user-defined window name (created using the WIN_CREATE built-in routine).
scrn_num indicates the screen which displays the window. It can be a user-created or system created screen. If not specified, SCRNU_USER is used.

Examples:

```

WIN_SEL ('menu:')
OPEN FILE lun ('CRT2:', 'RW')
WRITE lun ('Enter value: ')
WIN_SEL('CRT2:') -- on SCRNU_USER
READ lun (val)

```

11.184 WIN_SET_CRSR Built-In Procedure

The WIN_SET_CRSR built-in procedure places the cursor at a specified row and column on the window.

Calling Sequence: WIN_SET_CRSR (*row*, *col* <, *win_name*>)

Parameters: *row* : INTEGER [IN]
col : INTEGER [IN]
win_name : STRING [IN]

Comments: *row* and *col* specify the row and column position to which the cursor is to be set.
The home position (top, left corner) on a window is (0,0).
The optional parameter *win_name* can be used to specify the window. If it is not specified, the default window indicated by \$DFT_DV[1] is used. *win_name* can be one of the following system defined window names or any user-defined window name:

- ‘TP:’ -- scrolling window on system screen
- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on PC video system screen (WinC5G program is active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

Examples:

```

OPEN FILE crt1_lun ('CRT1:', 'RW')
row := 0
col := 5
WIN_SET_CRSR (row, col, 'CRT1:') -- sets to 0,5
WHILE row <= max_row DO
    WRITE (msg_str)
    row := row + 1
    col := 5
    WIN_SET_CRSR (row, col, 'CRT1:')
    WIN_GET_CRSR (row, col, 'CRT1:')
ENDWHILE

```

11.185 WIN_SIZE Built-In Procedure

The WIN_SIZE built-in procedure is used for dinamically change the size of a window according to the screen where this window should be displayed.

Note that the window must have been created with the largest size that will be used. When enlarging the window, sufficient space must be available on the screen or else the window will be truncated.

Calling Sequence: WIN_SIZE (<window_name>, <number_of_rows>)

Parameters: window_name : STRING [IN]
 number_of_rows : INTEGER [IN]

Comments: *window_name* can be one of the following system defined window names or any user-defined window name:

- ‘TP:’ -- scrolling window on system screen
- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on system screen of the PC video (when WinC5G program is active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

number_of_rows is the number of rows that the specified window must occupy.

Examples:

```

WIN_CREATE (win_name, PDV_TP, WIN_SCROLL, win_original_size)
IF screen_size < win_original_size THEN
    WIN_SIZE (win_name, win_reduced_size)
ENDIF
WIN_POPUP (win_name, 'TP2:')

```

11.186 WIN_STATE Built-In Function

The WIN_STATE built-in function returns the current state of the specified window.

Calling Sequence: WIN_STATE (win_name <, scrn_num <, parent <, num_rows>>)

Return Type: INTEGER

Parameters:

win_name : STRING	[IN]
scrn_num : INTEGER	[IN]
parent : STRING	[OUT]
num_rows : INTEGER	[OUT]

Comments: *win_name* can be one of the following system defined window names or any user-defined window name:

- ‘TP:’ -- scrolling window on system screen
- ‘TP1:’ -- window 1 on user screen on Teach Pendant
- ‘TP2:’ -- window 2 on user screen on Teach Pendant
- ‘TP3:’ -- window 3 on user screen on Teach Pendant
- ‘CRT:’ -- scrolling window on PC video system screen (when WinC5G program active)
- ‘CRT1:’ -- window 1 on user screen on PC video
- ‘CRT2:’ -- window 2 on user screen on PC video
- ‘CRT3:’ -- window 3 on user screen on PC video

The returned value is an INTEGER mask indicating the current window state. The meaning of the INTEGER is as follows:

- Bit 1 : is window displayed on SCRN_SYS
- Bit 2 : is window a popup on SCRN_SYS
- Bit 3 : is window displayed on SCRN_USER
- Bit 4 : is window a popup on SCRN_USER
- Bit 5 : are LUNs opened on window
- Bit 6 : is window attached to a program
- Bit 7 : is window defined for CRT
- Bits 8-10 : the foreground color
- Bits 11-13 : the background color
- Bit 14 : is window displayed on screen indicated by the *scrn_num* argument
- Bit 15 : is window a popup on screen indicated by the *scrn_num* argument
- Bit 16 : unused
- Bit 17 : is window saved
- Bit 18 : is input allowed on window
- Bit 19 : does output on window scroll
- Bits 20-21 : unused
- Bit 22 : is cursor turned off
- Bit 23 : unused
- Bit 24 : does output on window wrap
- Bits 25-32 : unused

scrn_num indicates the screen on which the window is to be displayed. It can be a user-created screen or a system created screen. System created screens are represented by the following predefined constants:

- SCRN_USER -- user screen
- SCRN_SYS -- system screen

parent will be set to the name of the window upon which *win_name* is popped up (if it isn't a fixed window).

num_rows will be set to the number of rows contained on *win_name*.

Examples:

```
-- Routine to write the state of the window
ROUTINE write_state(win : STRING; scrn : INTEGER)
    VAR state : INTEGER
        parent : STRING[6]
        nrows : INTEGER
    BEGIN
        state := WIN_STATE(win, scrn, parent, nrows)
        -- Write window name and state value
        WRITE (win::5, state::9::2)
        -- Device
        IF state AND ws_crt = ws_crt THEN
            WRITE (' CRT')
        ELSE
            WRITE (' TP')
        ENDIF
        ...
        -- Is there a parent ?
        IF NOT VAR_UNINIT(parent) THEN
            WRITE (parent::6)
        ELSE
            WRITE (' ::6)
        ENDIF
        -- Number of rows
        WRITE (nrows::3)
        -- File opened ?
        IF state AND ws_fopen = ws_fopen THEN
            WRITE ('X)::3)
        ELSE
            WRITE (' ::3)
        ENDIF
        ...
        IF state AND ws_nocrsr = ws_nocrsr THEN
            WRITE ('X)::3)
        ELSE
            WRITE (' ::3)
        ENDIF
        WRITE (NL)
    END write_state
```

12. PREDEFINED VARIABLES LIST

This chapter is a reference of PDL2 predefined variables. They are available in the following lists:

- [Predefined Variables groups](#)
- [Alphabetical Listing](#)

The following information is provided for each predefined variable:

- [Memory Category](#)
- [Load Category](#)
- [Data Type](#)
- [Limits](#) (“none” indicates no limits);
- [Attributes](#) (“none” indicates no special attributes);
- [S/W Version](#)
- [Unparsed](#)
- [Description](#).

Predefined variables begin with a dollar sign (\$) to easily distinguish them from user defined variables.

12.1 Memory Category

Predefined variables belong to one of the following memory categories which indicates where the value is located:

- Static: the predefined variable is shared between all PDL2 programs.
- Dynamic: these variables are dynamic allocated basing on the resources available in the system. For example, there is one \$ARM_DATA element for each arm declared in the controller.
- Port: they are INPUT/OUTPUT ports. Refer to [Ports](#) chapter for further details.
- Program Stack: there is one predefined variable of this category for each PDL2 program. The variable resides on the stack of the program that is created upon its activation.
- Program Loaded: it indicates those variables linked to each single program and, unlike Program Stack Category, they can be viewed when the program is just loaded and not active, by issuing a command with the /Property option (e.g. MemoryViewProgram/Property). If, for some reason, the Property output to a file (e.g. to a .LSV file) is not desired, it is sufficient to set bit 28 of \$CNTRL_CNFG to 1 for disabling the printing of the Property output (refer to [\\$CNTRL_CNFG: Controller configuration mode](#), bit 28, for further details).
- Field: the predefined variable is a field of the indicated predefined variable group. For example field of arm_data, field of crnt_data, ecc.

12.2 Load Category

Load Category classifies each system variable for loading and saving operations.

Predefined variables can be divided in the following major categories:

- Arm: these are arm dependent variables.
- Controller: these are controller related variables.
- Loop: these are variables related to the Loop configuration.
- Input/Output: there are Input/output dependent variables.
- Retentive: these variables are saved in NVRAM / FLASH memory and, upon restart, they are loaded with a higher precedence in respect with the .C5G file. Upon a Configure Save or Load they are also copied to the Retentive Memory, in order to align the content of the execution memory with the content of the Retentive Memory.
- Not saved: these variables are not saved in a .C5G file but are initialised by the system. Static and field predefined variable values can be saved and loaded using the ConfigureSave and ConfigureLoad commands as described in the C5G Use and Maintenance Manuals.



Note that complex structures such as \$ARM_DATA, \$LOOP_DATA are classified as “Not Saved” Load Category, even if some fields of theirs are always saved and loaded from .C5G file.

12.2.1 Minor Category

It is the sub-category which a predefined variable belongs to. It indicates which option can be applied during a ConfigureSaveCategory or a ConfigureLoadCategory command.

Example:

[\\$SENSOR_ENBL](#): Sensor Enable predefined variable belongs to ARM Load Category and to SENSOR Minor Category. Such a Minor Category will be referenced to, while using ConfigureLoadCategoryArm/Sensor and ConfigureSaveCategoryArm/Sensor commands.



For further information see also Control Unit Use Manual - Chap. SYSTEM COMMANDS.

12.3 Data Type

Predefined variables use the same data types as user defined variables, described in the [Data Representation](#) chapter, except for the system data types.

12.4 Attributes

The attributes section lists access information.

- Read-only. Usually, PDL2 programs can assign (write) a value to a predefined variable and can examine (read) the current value of a variable. The “read-only” attribute indicates that a predefined variable can only be read.
- WITH MOVE, WITH MOVE ALONG, WITH OPEN FILE: some predefined variables can assume a specific value that is assigned during a statement execution using the WITH clause.

- Limited Access. Some predefined variables can only be accessed using the WITH clause.
- Field node. The predefined variable can be used as a predefined field of a path node (NODEDEF data type definition).
- Pulse usable. The predefined variable can be used in a PULSE statement.
- Privileged read-write. Only COMAU technicians or COMAU programs can set this variables by means of a special mechanism.

12.5 Limits

This indication is present if the value of a predefined variable should be included in a specific range of values.

12.6 S/W Version

This field indicates in which system software version the predefined variable has been delivered.

12.7 Unparsed

If this information is present, it specifies the format (e.g. hexadecimal) in which the value of the predefined variable is shown when the configuration file (.C5G) is converted in an ASCII format (.PDL).

12.8 Predefined Variables groups

All predefined variables are classified in the following groups:

- PORT System Variables
- PROGRAM STACK System Variables
- PROGRAM LOADED System Variables
- ARM_DATA System Variables
- PROG_ARM_DATA System Variables
- CRNT_DATA System Variables
- INTERFERENCE REGIONS System Variables
- WEAVE_TBL System Variables
- ON_POS_TBL System Variables
- TRK_TBL System Variables
- WITH MOVE System Variables
- WITH MOVE ALONG System Variables
- WITH OPEN FILE System Variables
- PATH NODE FIELD System Variables
- IO_DEV System Variables
- IO_STS System Variables
- LOOP_DATA System Variables

- MISCELLANEOUS System Variables

12.8.1 PORT System Variables

- \$AIN: Analog input
- \$AOUT: Analog output
- \$BIT: BIT data
- \$CIO_TREEID: Tree identifier for the configure I/O tree
- \$DIN: Digital input
- \$DOUT: Digital output
- \$FDIN: Functional digital input
- \$FDOUT: Functional digital output
- \$FMI: Flexible Multiple Analog/Digital Inputs
- \$FMO: Flexible Multiple Analog/Digital Outputs
- \$GI: General System Input
- \$GO: General System Output
- \$HDIN: High speed digital input
- \$IN: IN digital
- \$OUT: OUT digital
- \$PROG_TIMER_O: Program timer - owning context specific (in ms)
- \$PROG_TIMER_OS: Program timer - owning context specific (in seconds)
- \$PROG_TIMER_X: Program timer - execution context specific (in ms)
- \$PROG_TIMER_XS: Program timer - execution context specific (in seconds)
- \$SAFE_SPD: User Velocity for testing safety speed
- \$SDO: System digital output
- \$TIMER: Clock timer (in ms)
- \$TIMER_S: Clock timer (in seconds)
- \$WORD: WORD data

12.8.2 PROGRAM STACK System Variables

- \$CYCLE: Program cycle count
- \$ERROR: Last PDL2 Program Error
- \$ERROR_DATA: Last PDL2 Program Error Data
- \$FLY_PER: Percentage of fly motion
- \$FLY_TRAJ: Type of control on cartesian fly
- \$FLY_TYPE: Type of fly motion
- \$FL_STS: Status of last file operation
- \$MOVE_TYPE: Type of motion
- \$ORNT_TYPE: Type of orientation
- \$PROG_ACC_OVR: Program acceleration override
- \$PROG_ARG: Program's activation argument

- \$PROG_ARM: Arm of program
- \$PROG_BREG: Boolean Registers - Program specific
- \$PROG_CNFG: Program configuration
- \$PROG_CONDS: Defined conditions of a program
- \$PROG_DEC_OVR: Program deceleration override
- \$PROG_IREG: Integer Registers - Program specific
- \$PROG_LINE: executing program line
- \$PROG_LUN: program specific default LUN
- \$PROG_NAME: Executing program name
- \$PROG_OWNER: Program Owner of executing line
- \$PROG_RES: Program's execution result
- \$PROG_RREG: Real Registers - Program specific
- \$PROG_SPD_OVR: Program speed override
- \$PROG_SREG: String Registers - Program specific
- \$PROG_TIMER_O: Program timer - owning context specific (in ms)
- \$PROG_TIMER_OS: Program timer - owning context specific (in seconds)
- \$PROG_TIMER_X: Program timer - execution context specific (in ms)
- \$PROG_TIMER_XS: Program timer - execution context specific (in seconds)
- \$READ_TOUT: Timeout on a READ
- \$SPD_OPT: Type of speed control
- \$STRESS_PER: Stress percentage in cartesian fly
- \$SYNC_ARM: Synchronized arm of program
- \$SYS_CALL_OUT: Output lun for SYS_CALL
- \$SYS_CALL_STS: Status of last SYS_CALL
- \$SYS_CALL_TOUT: Timeout for SYS_CALL
- \$TERM_TYPE: Type of motion termination
- \$THRD_CEXP: Thread Condition Expression
- \$THRD_ERROR: Error of each thread of execution
- \$THRD_PARAM: Thread Parameter
- \$WEAVE_MODALITY: Weave modality
- \$WEAVE_NUM: Weave table number
- \$WEAVE_TYPE: Weave type
- \$WFR_IOTOUT: Timeout on a WAIT FOR when IO simulated
- \$WFR_TOUT: Timeout on a WAIT FOR
- \$WRITE_TOUT: Timeout on a WRITE

12.8.3 PROGRAM LOADED System Variables

- \$PROP_AUTHOR: last Author who saved the file
- \$PROP_DATE: date and time when the program was last saved

- \$PROP_FILE: property information for loaded file
- \$PROP_HELP: the help the user wants
- \$PROP_HOST: Controller ID or PC user domain, upon which the file was last saved
- \$PROP_REVISION: user defined string representing the version
- \$PROP_TITLE: the title defined by the user for the file
- \$PROP_UML: user modify level
- \$PROP_UVL: user view level
- \$PROP_VERSION: version upon which it was built

12.8.4 ARM_DATA System Variables

- \$ARM_ACC_OVR: Arm acceleration override
- \$ARM_DATA: Robot arm data
- \$ARM_DEC_OVR: Arm deceleration override
- \$ARM_LINKED: Enable/disable arm coupling
- \$ARM_OVR: Arm override
- \$ARM_SENSITIVITY: Arm collision sensitivity
- \$ARM_SPD_OVR: Arm speed override
- \$AUX_BASE: Auxiliary base for a positioner of an arm
- \$AUX_KEY: Teach Pendant AUX-A and AUX-B keys mapping
- \$AUX_MASK: Auxiliary arm mask
- \$AUX_OFST: Auxiliary axes offsets
- \$AUX_TYPE: Positioner type
- \$AX_CNVRSN: Axes conversion
- \$AX_INF: Axes inference
- \$AX_LEN: Axes lengths
- \$AX_OFST: Axes offsets
- \$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature
- \$AX_PURSUIT_ENBL: Axes Pursuit enabling flag
- \$AX_PURSUIT_LINKED: Axes Pursuit linked
- \$A_ALONG_1D: Internal arm data
- \$A_ALONG_2D: Internal arm data
- \$A_AREAL_1D: Internal arm data
- \$A_AREAL_2D: Internal arm data
- \$BASE: Base of arm
- \$CAL_DATA: Calibration data
- \$CAL_SYS: System calibration position
- \$CAL_USER: User calibration position
- \$CNFG_CARE: Configuration care

- \$COLL_EFFECT: Collision Effect on the arm status
- \$COLL_SOFT_PER: Collision compliance percentage
- \$COLL_TYPE: Type of collision
- \$DYN_COLL_FILTER: Dynamic Collision Filter
- \$DYN_DELAY: Dynamic model delay
- \$DYN_FILTER2: Dynamic Filter for dynamic model
- \$DYN_GAIN: Dynamic gain in inertia and viscous friction
- \$DYN_MODEL: Dynamic Model
- \$DYN_WRIST: Dynamic Wrist
- \$DYN_WRISTQS: Dynamic Theta
- \$FLY_DIST: Distance in fly motion
- \$FL_COMP: Compensation file name
- \$GUN: Electrical welding gun
- \$HAND_TYPE: Type of hand
- \$HLD_DEC_PER: Hold deceleration percentage
- \$HOME: Arm home position
- \$JERK: Jerk control values
- \$JL_TABLE: Internal Arm data
- \$JNT_LIMIT_AREA: Joint limits of the work area
- \$JNT_MASK: Joint arm mask
- \$JNT_MTURN: Check joint Multi-turn
- \$JNT_OVR: joint override
- \$JNT_SM: joint trajectory planning indicator
- \$JOG_INCR_DIST: Increment Jog distance
- \$JOG_INCR_ROT: Rotational jog increment
- \$JOG_SPD_OVR: Jog speed override
- \$LIN_ACC_LIM: Linear acceleration limit
- \$LIN_DEC_LIM: Linear deceleration limit
- \$LIN_SPD: Linear speed
- \$LIN_SPD_LIM: Linear speed limit
- \$LIN_SPD_RT: Run-time Linear speed
- \$LIN_SPD_RT_OVR: Run-time Linear speed override
- \$LOOP_DATA: Loop data
- \$MAN_SCALE: Manual scale factor
- \$MOD_ACC_DEC: Modulation of acceleration and deceleration
- \$MOD_MASK: Joint mod mask
- \$MTR_ACC_TIME: Motor acceleration time
- \$MTR_DEC_TIME: Motor deceleration time
- \$MTR_SPD_LIM: Motor speed limit

- \$NUM_AUX_AXES: Number of auxiliary axes
- \$NUM_JNT_AXES: Number of joint axes
- \$OP_TOL_DIST: On Pos-Jnt Tolerance distance
- \$OP_TOL_ORNT: On Pos-Jnt Tolerance Orientation
- \$OT_TOL_DIST: On Trajectory Tolerance distance
- \$OT_TOL_ORNT: On Trajectory Orientation
- \$PAR: Nodal motion variable
- \$PGOV_ACCURACY: required accuracy in cartesian motions
- \$PGOV_MAX_SPD_REDUCTION: Maximum speed scale factor
- \$PGOV_ORNT_PER: percentage of orientation
- \$POS_LIMIT_AREA: Cartesian limits of work area
- \$RB_FAMILY: Family of the robot arm
- \$RB_MODEL: Model of the robot arm
- \$RB_NAME: Name of the robot arm
- \$RB_STATE: State of the robot arm
- \$RB_VARIANT: Variant of the robot arm
- \$RCVR_DIST: Distance from the recovery position
- \$RCVR_TYPE: Type of motion recovery
- \$ROT_ACC_LIM: Rotational acceleration limit
- \$ROT_DEC_LIM: Rotational deceleration limit
- \$ROT_SPD: Rotational speed
- \$ROT_SPD_LIM: Rotational speed limit
- \$SFRAME: Sensor frame of an arm
- \$SING_CARE: Singularity care
- \$SM4C_SAT_VEL_SCALE: thresholds for the joint speed saturation in Cartesian SmartMove
- \$SM4C_STRESS_PER: Maximum Stress allowed in Cartesian SmartMove
- \$SM4_SAT_ADD_SCALE: Additional threshold for the SmartMove saturation
- \$STRK_END_N: User negative stroke end
- \$STRK_END_P: User positive stroke end
- \$STRK_END_SYS_N: System stroke ends
- \$STRK_END_SYS_P: System stroke ends
- \$TOL_ABST: Tolerance anti-bouce time
- \$TOL_COARSE: Tolerance coarse
- \$TOL_FINE: Tolerance fine
- \$TOL_JNT_COARSE: Tolerance for joints
- \$TOL_JNT_FINE: Tolerance for joints
- \$TOL_TOUT: Tolerance timeout
- \$TOOL: Tool of arm

- \$TOOL_CNTR: Tool center of mass of the tool
- \$TOOL_INERTIA: Tool Inertia
- \$TOOL_MASS: Mass of the tool
- \$TOOL_XTREME: Extreme Tool of the Arm
- \$TOOL_RMT: Fixed Tool
- \$TP_ORNT: Orientation for jog motion
- \$TURN_CARE: Turn care
- \$TX_RATE: Transmission rate
- \$UFRAME: User frame of an arm

12.8.5 PROG_ARM_DATA System Variables

- \$PROG_ARM_DATA: Arm-related Data - Program specific

12.8.6 CRNT_DATA System Variables

- \$ARM_DISB: Arm disable flags
- \$ARM_SIMU: Arm simulate flag
- \$ARM_SPACE: current Arm Space
- \$ARM_VEL: Arm velocity
- \$AX_PURSUIT_ENBL: Axes Pursuit enabling flag
- \$CAUX_POS: Cartesian positioner position
- \$COLL_ENBL: Collision enabling flag
- \$CONV_ACT: Conveyor Act
- \$CONV_BEND_ANGLE: Conveyor Bend Angle
- \$CONV_DIST: Conveyor shift in micron
- \$CONV_ENC: Conveyor Encoder Position
- \$CONV_SHIFT: Conveyor shift in mm
- \$CONV_SPD: Conveyor Speed
- \$C_ALONG_1D: Internal current arm data
- \$C_AREAL_1D: Internal current arm data
- \$C_AREAL_2D: Internal current arm data
- \$FLY_DBUG: Cartesian Fly Debug
- \$FOLL_ERR: Following error
- \$JOG_INCR_ENBL: Jog incremental motion
- \$MOVE_STATE: Move state
- \$MTR_CURR: Motor current
- \$ODO_METER: average TCP space
- \$OT_COARSE: On Trajectory indicator
- \$OT_JNT: On Trajectory joint position
- \$OT_POS: On Trajectory position
- \$OT_TOOL: On Trajectory TOOL position

- \$OT_TOOL_RMT: On Trajectory remote tool flag
- \$OT_UFRAME: On Trajectory User frame
- \$OT_UNINIT: On Trajectory position uninit flag
- \$RAD_IDL_QUO: Radian ideal quote
- \$RAD_TARG: Radian target
- \$RAD_VEL: Radian velocity
- \$RCVR_LOCK: Change arm state after recovery
- \$SAFE_ENBL: Safe speed enabled
- \$WEAVE_MODALITY_NOMOT: Weave modality (only for no arm motion)
- \$WEAVE_NUM_NOMOT: Weave table number (only for no arm motion)
- \$WEAVE_PHASE: Index of the Weaving Phase
- \$WEAVE_TYPE_NOMOT: Weave type (only for no arm motion)

12.8.7 INTERFERENCE REGIONS System Variables

- \$IR_ARM: Interference Region Arm
- \$IR_CNFG: Interference Region Configuration
- \$IR_ENBL: Interference Region enabled flag
- \$IR_IN_OUT: Interference Region location flag
- \$IR_JNT_N: Interference Region negative joint
- \$IR_JNT_P: Interference Region positive joint
- \$IR_PBIT: Interference Region port bit
- \$IR_PERMANENT: Interference Region permanent
- \$IR_PIDX: Interference Region port index
- \$IR_POS1: Interference Region position
- \$IR_POS2: Interference Region position
- \$IR_POS3: Interference Region position
- \$IR_POS4: Interference Region position
- \$IR_PTYPE: Interference Region port type
- \$IR_REACHED: Interference Region position reached flag
- \$IR_SHAPE: Interference Region Shape
- \$IR_TBL: Interference Region table data
- \$IR_TYPE: Interference Region type
- \$IR_UFRAME: Interference Region Uframe

12.8.8 WEAVE_TBL System Variables

- \$WV_AMPLITUDE: Weave amplitude
- \$WV_AMP_PER_RIGHT/LEFT: Weave amplitude percentage
- \$WV_CNTR_DWL: Weave center dwell
- \$WV_DIRECTION_ANGLE: Weave angle direction
- \$WV_END_DWL: Weave end dwell

- \$WV_LEFT_AMP: Weave left amplitude
- \$WV_LEFT_DWL: Weave left dwell
- \$WV_ONE_CYCLE: Weave one cycle
- \$WV_PLANE: Weave plane angle
- \$WV_PLANE_MODALITY: Weave plane modality
- \$WV_RADIUS: Weave radius
- \$WV_RIGHT_AMP: Weave right amplitude
- \$WV_RIGHT_DWL: Weave right dwell
- \$WV_SMOOTH: Weave smooth enabled
- \$WV_SPD_PROFILE: Weave speed profile enabled
- \$WV_TRV_SPD: Weave transverse speed
- \$WV_TRV_SPD_PHASE: Weave transverse speed phase

12.8.9 ON_POS_TBL System Variables

- \$OP_JNT: On Pos jointpos
- \$OP_JNT_MASK: On Pos Joint Mask
- \$OP_POS: On Pos position
- \$OP_REACHED: On Pos position reached flag
- \$OP_TOOL: The On Pos Tool
- \$OP_TOOL_DSBL: On Pos tool disable flag
- \$OP_TOOL_RMT: On Pos Remote tool flag
- \$OP_UFRAME: The On Pos Uframe

12.8.10 TRK_TBL System Variables

- \$TRK_TBL: Tracking application table data
- \$TT_APPL_ID: Tracking application identifier
- \$TT_ARM_MASK: Tracking arm mask
- \$TT_FRAMES: Array of POSITIONs for tracking application
- \$TT_I_PRMS: Integer parameters for the tracking application
- \$TT_PORT_IDX: Port Index for the tracking application
- \$TT_PORT_TYPE: Port Type for the tracking application
- \$TT_R_PRMS: Real parameters for the tracking application

12.8.11 WITH MOVE System Variables

- \$ARM_ACC_OVR: Arm acceleration override
- \$ARM_DEC_OVR: Arm deceleration override
- \$ARM_LINKED: Enable/disable arm coupling
- \$ARM_SENSITIVITY: Arm collision sensitivity
- \$ARM_SPD_OVR: Arm speed override
- \$AUX_OFST: Auxiliary axes offsets

- \$BASE: Base of arm
- \$CNFG_CARE: Configuration care
- \$COLL_SOFT_PER: Collision compliance percentage
- \$COLL_TYPE: Type of collision
- \$FLY_DIST: Distance in fly motion
- \$FLY_PER: Percentage of fly motion
- \$FLY_TRAJ: Type of control on cartesian fly
- \$FLY_TYPE: Type of fly motion
- \$JNT_MTURN: Check joint Multi-turn
- \$JNT_OVR: joint override
- \$LIN_SPD: Linear speed
- \$LIN_SPD_RT: Run-time Linear speed
- \$MOVE_TYPE: Type of motion
- \$ORNT_TYPE: Type of orientation
- \$PAR: Nodal motion variable
- \$PROG_ACC_OVR: Program acceleration override
- \$PROG_DEC_OVR: Program deceleration override
- \$PROG_SPD_OVR: Program speed override
- \$ROT_SPD: Rotational speed
- \$SFRAME: Sensor frame of an arm
- \$SING_CARE: Singularity care
- \$SPD_OPT: Type of speed control
- \$STRESS_PER: Stress percentage in cartesian fly
- \$TERM_TYPE: Type of motion termination
- \$TOL_COARSE: Tolerance coarse
- \$TOL_FINE: Tolerance fine
- \$TOOL: Tool of arm
- \$TOOL_CNTR: Tool center of mass of the tool
- \$TOOL_INERTIA: Tool Inertia
- \$TOOL_MASS: Mass of the tool
- \$TOOL_RMT: Fixed Tool
- \$TURN_CARE: Turn care
- \$UFRAME: User frame of an arm
- \$WEAVE_NUM: Weave table number
- \$WEAVE_TYPE: Weave type
- \$WV_AMP_PER_RIGHT/LEFT: Weave amplitude percentage

12.8.12 WITH MOVE ALONG System Variables

- \$ARM_ACC_OVR: Arm acceleration override

- \$ARM_DEC_OVR: Arm deceleration override
- \$ARM_SPD_OVR: Arm speed override
- \$AUX_OFST: Auxiliary axes offsets
- \$BASE: Base of arm
- \$CNFG_CARE: Configuration care
- \$COLL_TYPE: Type of collision
- \$COND_MASK: PATH segment condition mask
- \$FLY_DIST: Distance in fly motion
- \$FLY_PER: Percentage of fly motion
- \$FLY_TRAJ: Type of control on cartesian fly
- \$FLY_TYPE: Type of fly motion
- \$JNT_MTURN: Check joint Multi-turn
- \$JNT_OVR: joint override
- \$LIN_SPD: Linear speed
- \$LIN_SPD_RT: Run-time Linear speed
- \$MOVE_TYPE: Type of motion
- \$ORNT_TYPE: Type of orientation
- \$ROT_SPD: Rotational speed
- \$SEG_DATA: PATH segment data
- \$SEG_FLY: PATH segment fly or not
- \$SEG_FLY_DIST: Parameter in segment fly motion
- \$SEG_FLY_PER: PATH segment fly percentage
- \$SEG_FLY_TRAJ: Type of fly control
- \$SEG_FLY_TYPE: PATH segment fly type
- \$SEG_OVR: PATH segment override
- \$SEG_REF_IDX: PATH segment reference index
- \$SEG_STRESS_PER: Percentage of stress required in fly
- \$SEG_TERM_TYPE: PATH segment termination type
- \$SEG_TOL: PATH segment tolerance
- \$SEG_TOOL_IDX: PATH segment tool index
- \$SEG_WAIT: PATH segment WAIT
- \$SING_CARE: Singularity care
- \$SPD_OPT: Type of speed control
- \$STRESS_PER: Stress percentage in cartesian fly
- \$TERM_TYPE: Type of motion termination
- \$TOL_COARSE: Tolerance coarse
- \$TOL_FINE: Tolerance fine
- \$TOOL: Tool of arm
- \$TOOL_CNTR: Tool center of mass of the tool

- \$TOOL_MASS: Mass of the tool
- \$TURN_CARE: Turn care
- \$WEAVE_NUM: Weave table number
- \$WEAVE_TYPE: Weave type

12.8.13 WITH OPEN FILE System Variables

- \$FL_ADLMT: Array of delimiters
- \$FL_BINARY: Text or character mode
- \$FL_DLMT: Delimiter specification
- \$FL_ECHO: Echo characters
- \$FL_NUM_CHARS: Number of chars to be read
- \$FL_PASSALL: Pass all characters
- \$FL_RANDOM: Random file access
- \$FL_RDFLUSH: Flush on reading
- \$FL_SWAP: Low or high byte first

12.8.14 PATH NODE FIELD System Variables

- \$CNFG_CARE: Configuration care
- \$COND_MASK: PATH segment condition mask
- \$COND_MASK_BACK: PATH segment condition mask in backwards
- \$JNT_MTURN: Check joint Multi-turn
- \$LIN_SPD: Linear speed
- \$MAIN_JNTP: PATH node main jointpos destination
- \$MAIN_POS: PATH node main position destination
- \$MAIN_XTND: PATH node main xtndpos destination
- \$MOVE_TYPE: Type of motion
- \$ORNT_TYPE: Type of orientation
- \$ROT_SPD: Rotational speed
- \$SEG_DATA: PATH segment data
- \$SEG_FLY: PATH segment fly or not
- \$SEG_FLY_DIST: Parameter in segment fly motion
- \$SEG_FLY_PER: PATH segment fly percentage
- \$SEG_FLY_TRAJ: Type of fly control
- \$SEG_FLY_TYPE: PATH segment fly type
- \$SEG_OVR: PATH segment override
- \$SEG_REF_IDX: PATH segment reference index
- \$SEG_STRESS_PER: Percentage of stress required in fly
- \$SEG_TERM_TYPE: PATH segment termination type
- \$SEG_TOL: PATH segment tolerance
- \$SEG_TOOL_IDX: PATH segment tool index

- \$SEG_WAIT: PATH segment WAIT
- \$SING_CARE: Singularity care
- \$SPD_OPT: Type of speed control
- \$TURN_CARE: Turn care
- \$WEAVE_NUM: Weave table number
- \$WEAVE_TYPE: Weave type

12.8.15 IO_DEV System Variables

- \$ID_ADRS: Address on the Fieldbus net
- \$ID_APPL: Application Code
- \$ID_DATA: Fieldbus specific data
- \$ID_ERR: error Code
- \$ID_IDX: Device index
- \$ID_ISIZE: Input bytes quantity
- \$ID_ISMST: this Device is a Master (or Controller)
- \$ID_MASK: miscellaneous mask of bits
- \$ID_NAME: Device name
- \$ID_NET: Fieldbus type
- \$ID_NOT_ACTIVE: not active Device at net boot time
- \$ID_OSIZE: output bytes quantity
- \$ID_STS: Device status
- \$ID_TYPE: configuration status

12.8.16 IO_STS System Variables

- \$IS_DEV: associated Device
- \$IS_HELP_INT: User Help code
- \$IS_HELP_STR: User Help string
- \$IS_NAME: I/O point name
- \$IS_PORT_CODE: Port type
- \$IS_PORT_INDEX: Port type index
- \$IS_PRIV: privileged Port flag
- \$IS_RTN: retentive output flag
- \$IS_SIZE: Port dimension
- \$IS_STS: Port status

12.8.17 LOOP_DATA System Variables

- \$LOOP_DATA: Loop data
- \$L_ALONG_1D: Internal Loop data
- \$L_ALONG_2D: Internal Loop data
- \$L_AREAL_1D: Internal Loop data

- \$L_AREAL_2D: Internal Loop data

12.8.18 MISCELLANEOUS System Variables

- \$APPL_ID: Application Identifier
- \$APPL_NAME: Application Identifiers
- \$APPL_OPTIONS: Application Options
- \$ARM_DATA: Robot arm data
- \$ARM_USED: Program use of arms
- \$BACKUP_SET: Default devices
- \$BOOTLINES: Bootline read-only
- \$BREG: Boolean registers - saved
- \$BREG_NS: Boolean registers - not saved
- \$B_ASTR_1D_NS: Board string data
- \$B_ALONG_1D_NS: Internal board data
- \$B_NVRAM: NVRAM data of the board
- \$CNTRL_CNFG: Controller configuration mode
- \$CNTRL_DST: Controller Day Light Saving
- \$CNTRL_INIT: Controller initialization mode
- \$CNTRL_OPTIONS: Controller Options
- \$CNTRL_TZ: Controller Time Zone
- \$CRC_IMPORT: Directory for IMPORT
- \$CRC_RULES: C5G Save & Load rules
- \$CRNT_DATA: Current Arm data
- \$DB_MSG: Message Database Identifier
- \$DEPEND_DIRS: Dependency path
- \$DFT_ARM: Default arm
- \$DFT_DV: Default devices
- \$DFT_LUN: Default LUN number
- \$DFT_SPD: Default devices speed
- \$DNS_DOMAIN: DNS Domain
- \$DNS_ORDER: DNS Order
- \$DNS_SERVERS: DNS Servers
- \$DV_STS: the status of DV_CNTRL calls
- \$DV_TOUT: Timeout for asynchronous DV_CNTRL calls
- \$EMAIL_INT: Email integer configuration:
- \$EMAIL_STR: Email string configuration:
- \$EXE_HELP: Help on Execute command
- \$FEED_VOLTAGE: Feed Voltage
- \$FLOW_TBL: Flow modulation algorithm table data

- \$FL_ADLM: Array of delimiters
- \$FL_BINARY: Text or character mode
- \$FL_CNFG: Configuration file name
- \$FL_DLMT: Delimiter specification
- \$FL_ECHO: Echo characters
- \$FL_NUM_CHARS: Number of chars to be read
- \$FL_PASSALL: Pass all characters
- \$FL_RANDOM: Random file access
- \$FL_RDFLUSH: Flush on reading
- \$FL_SWAP: Low or high byte first
- \$FUI_DIRS: Installation path
- \$FW_ARM: Arm under flow modulation algorithm
- \$FW_AXIS: Axis under flow modulation algorithm
- \$FW_CNVRSN: Conversion factor in case of Flow modulation algorithm
- \$FW_ENBL Flow modulation algorithm enabling indicator
- \$FW_FLOW_LIM Flow modulation algorithm flow limit
- \$FW_SPD_LIM Flow modulation algorithm speed limits
- \$FW_START Delay in flow modulation algorithm application after start
- \$FW_VAR: flag defining the variable to be considered when flow modulate is used
- \$GEN_OVR: General override
- \$IREG: Integer register - saved
- \$IREG_NS: Integer registers - not saved
- \$JREG: Jointpos registers - saved
- \$JREG_NS: Jointpos register - not saved
- \$LATCH_CNF: Latched alarm configuration setting
- \$MDM_INT: Modem Configuration
- \$MDM_STR: Modem Configuration:
- \$NET_B: Ethernet Boot Setup
- \$NET_B_DIR: Ethernet Boot Setup Directory
- \$NET_C_CNF: Ethernet Client Setting Modes
- \$NET_C_DIR: Ethernet Client Setup Default Directory
- \$NET_C_HOST: Ethernet Client Setup Remote Host
- \$NET_C_PASS: Ethernet Client Setup Password
- \$NET_C_USER: Ethernet Client Setup Login Name
- \$NET_HOSTNAME: Ethernet network hostnames
- \$NET_I_INT: Ethernet Network Information (integers)
- \$NET_I_STR: Ethernet Network Information (strings)
- \$NET_L: Ethernet Local Setup
- \$NET_MOUNT: Ethernet network mount

- \$NET_R_STR: Ethernet Remote Interface Setup
- \$NET_Q_STR: Ethernet Remote Interface Information:
- \$NET_S_INT: Ethernet Network Server Setup
- \$NET_T_INT: Ethernet Network Timer
- \$NET_T_HOST: Ethernet Network Time Protocol Host
- \$NOLOG_ERROR: Exclude messages from logging
- \$NUM_ALOG_FILES: Number of action log files
- \$NUM_ARMS: Number of arms
- \$NUM_DBs: Number of allowed Databases
- \$NUM_DEVICES: Number of Devices
- \$NUM_LUNS: Number of LUNs
- \$NUM_MB: Number of motion buffers
- \$NUM_MB_AHEAD: Number of motion buffers ahead
- \$NUM_PROGS: Number of active programs
- \$NUM_SCRNS: Number of screens
- \$NUM_TIMERS: Number of timers
- \$NUM_VP2_SCRNS: Number of Visual PDL2 screens
- \$NUM_WEAVES: Number of weaves (WEAVE_TBL)
- \$ON_POS_TBL: ON POS table data
- \$PREG: Position registers - saved
- \$PREG_NS: Position registers - not saved
- \$PWR_RCVR: Power failure recovery mode
- \$RBT_CNFG: Robot board configuration
- \$REC_SETUP: RECord key setup
- \$REMOTE: Functionality of the key in remote
- \$REM_I_STR: Remote connections Information
- \$REM_TUNE: Internal remote connection tuning parameters
- \$RESTART: Restart Program
- \$RESTART_MODE: Restart mode
- \$RESTORE_SET: Default devices
- \$RREG: Real registers - saved
- \$RREG_NS: Real registers - not saved
- \$SERIAL_NUM: Serial Number of the board
- \$SREG: String registers - saved
- \$SREG_NS: String registers - not saved
- \$STARTUP: Startup program
- \$STARTUP_USER: Configuration file name
- \$SYS_ERROR: Last system error
- \$SYS_ID: Robot System identifier

- \$SYS_PARAMS: Robot system identifier
- \$SYS_STATE: State of the system
- \$TP_ARM: Teach Pendant current arm
- \$TP_GEN_INCR: Incremental value for general override
- \$TP_MJOG: Type of TP jog motion
- \$TP_SYNC_ARM: Teach Pendant's synchronized arms
- \$TUNE: Internal tuning parameters
- \$VERSION: Software version
- \$VP2_SCRN_ID: Executing program VP2 Screen Identifier
- \$VP2_TOUT: Timeout value for asynchronous VP2 requests
- \$VP2_TUNE: Visual PDL2 tuning parameters
- \$WEAVE_TBL: Weave table data
- \$XREG: Xtndpos registers - saved
- \$XREG_NS: Xtndpos registers - not saved

12.9 Alphabetical Listing

- \$A_ALONG_1D: Internal arm data
- \$A_ALONG_2D: Internal arm data
- \$A_AREAL_1D: Internal arm data
- \$A_AREAL_2D: Internal arm data
- \$AIN: Analog input
- \$AOUT: Analog output
- \$APPL_ID: Application Identifier
- \$APPL_NAME: Application Identifiers
- \$APPL_OPTIONS: Application Options
- \$ARM_ACC_OVR: Arm acceleration override
- \$ARM_DATA: Robot arm data
- \$ARM_DEC_OVR: Arm deceleration override
- \$ARM_DISB: Arm disable flags
- \$ARM_LINKED: Enable/disable arm coupling
- \$ARM_OVR: Arm override
- \$ARM_SENSITIVITY: Arm collision sensitivity
- \$ARM_SIMU: Arm simulate flag
- \$ARM_SPACE: current Arm Space
- \$ARM_SPD_OVR: Arm speed override
- \$ARM_USED: Program use of arms
- \$ARM_VEL: Arm velocity
- \$AUX_BASE: Auxiliary base for a positioner of an arm

- \$AUX_KEY: Teach Pendant AUX-A and AUX-B keys mapping
- \$AUX_MASK: Auxiliary arm mask
- \$AUX_OFST: Auxiliary axes offsets
- \$AUX_TYPE: Positioner type
- \$AX_CNVRSN: Axes conversion
- \$AX_INF: Axes inference
- \$AX_LEN: Axes lengths
- \$AX_OFST: Axes offsets
- \$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature
- \$AX_PURSUIT_ENBL: Axes Pursuit enabling flag
- \$AX_PURSUIT_LINKED: Axes Pursuit linked
- \$B_ALONG_1D_NS: Internal board data
- \$B_ASTR_1D_NS: Board string data
- \$B_NVRAM: NVRAM data of the board
- \$BACKUP_SET: Default devices
- \$BASE: Base of arm
- \$BIT: BIT data
- \$BOOTLINES: Bootline read-only
- \$BREG: Boolean registers - saved
- \$BREG_NS: Boolean registers - not saved
- \$C_ALONG_1D: Internal current arm data
- \$C_AREAL_1D: Internal current arm data
- \$C_AREAL_2D: Internal current arm data
- \$CAL_DATA: Calibration data
- \$CAL_SYS: System calibration position
- \$CAL_USER: User calibration position
- \$CAUX_POS: Cartesian positioner position
- \$CIO_TREEID: Tree identifier for the configure I/O tree
- \$CNFG_CARE: Configuration care
- \$CNTRL_CNFG: Controller configuration mode
- \$CNTRL_DST: Controller Day Light Saving
- \$CNTRL_INIT: Controller initialization mode
- \$CNTRL_OPTIONS: Controller Options
- \$CNTRL_TZ: Controller Time Zone
- \$COLL_EFFECT: Collision Effect on the arm status
- \$COLL_ENBL: Collision enabling flag
- \$COLL_SOFT_PER: Collision compliance percentage
- \$COLL_TYPE: Type of collision
- \$COND_MASK: PATH segment condition mask

- \$COND_MASK_BACK: PATH segment condition mask in backwards
- \$CONV_ACT: Conveyor Act
- \$CONV_BEND_ANGLE: Conveyor Bend Angle
- \$CONV_DIST: Conveyor shift in micron
- \$CONV_ENC: Conveyor Encoder Position
- \$CONV_SHIFT: Conveyor shift in mm
- \$CONV_SPD: Conveyor Speed
- \$CRC_IMPORT: Directory for IMPORT
- \$CRC_RULES: C5G Save & Load rules
- \$CRNT_DATA: Current Arm data
- \$CUSTOM_CNTRL_ID: Identifier for the Controller
- \$CUSTOM_ID: Identifier for the arm
- \$CYCLE: Program cycle count
- \$DB_MSG: Message Database Identifier
- \$DELTPOS: offsets on final position in the specified (\$DELTPOS_IN) frame
- \$DELTPOS_IN: X, Y, Z offsets on final position
- \$DEPEND_DIRS: Dependency path
- \$DFT_ARM: Default arm
- \$DFT_DV: Default devices
- \$DFT_LUN: Default LUN number
- \$DFT_SPD: Default devices speed
- \$DIN: Digital input
- \$DNS_DOMAIN: DNS Domain
- \$DNS_ORDER: DNS Order
- \$DNS_SERVERS: DNS Servers
- \$DOUT: Digital output
- \$DV_STS: the status of DV_CNTRL calls
- \$DV_TOUT: Timeout for asynchronous DV_CNTRL calls
- \$DYN_COLL_FILTER: Dynamic Collision Filter
- \$DYN_DELAY: Dynamic model delay
- \$DYN_FILTER2: Dynamic Filter for dynamic model
- \$DYN_GAIN: Dynamic gain in inertia and viscous friction
- \$DYN_MODEL: Dynamic Model
- \$DYN_WRIST: Dynamic Wrist
- \$DYN_Wristqs: Dynamic Theta
- \$EMAIL_INT: Email integer configuration
- \$EMAIL_STR: Email string configuration
- \$ERROR: Last PDL2 Program Error
- \$ERROR_DATA: Last PDL2 Program Error Data

- \$EXE_HELP: Help on Execute command
- \$FDIN: Functional digital input
- \$FDOUT: Functional digital output
- \$FEED_VOLTAGE: Feed Voltage
- \$FL_ADLMT: Array of delimiters
- \$FL_ARM_CNFG: Arm configuration file name
- \$FL_AX_CNFG: Axes configuration file name
- \$FL_BINARY: Text or character mode
- \$FL_CNFG: Configuration file name
- \$FL_COMP: Compensation file name
- \$FL_DLMT: Delimiter specification
- \$FL_ECHO: Echo characters
- \$FL_NUM_CHARS: Number of chars to be read
- \$FL_PASSALL: Pass all characters
- \$FL_RANDOM: Random file access
- \$FL_RDFLUSH: Flush on reading
- \$FL_STS: Status of last file operation
- \$FL_SWAP: Low or high byte first
- \$FLOW_TBL: Flow modulation algorithm table data
- \$FLY_DBUG: Cartesian Fly Debug
- \$FLY_DIST: Distance in fly motion
- \$FLY_PER: Percentage of fly motion
- \$FLY_TRAJ: Type of control on cartesian fly
- \$FLY_TYPE: Type of fly motion
- \$FMI: Flexible Multiple Analog/Digital Inputs
- \$FMO: Flexible Multiple Analog/Digital Outputs
- \$FOLL_ERR: Following error
- \$FUI_DIRS: Installation path
- \$FW_ARM: Arm under flow modulation algorithm
- \$FW_AXIS: Axis under flow modulation algorithm
- \$FW_CNVRSN: Conversion factor in case of Flow modulation algorithm
- \$FW_ENBL Flow modulation algorithm enabling indicator
- \$FW_FLOW_LIM Flow modulation algorithm flow limit
- \$FW_SPD_LIM Flow modulation algorithm speed limits
- \$FW_START Delay in flow modulation algorithm application after start
- \$FW_VAR: flag defining the variable to be considered when flow modulate is used
- \$GEN_OVR: General override
- \$GI: General System Input
- \$GO: General System Output

- \$GUN: Electrical welding gun
- \$HAND_TYPE: Type of hand
- \$HDIN: High speed digital input
- \$HLD_DEC_PER: Hold deceleration percentage
- \$HOME: Arm home position
- \$ID_ADRS: Address on the Fieldbus net
- \$ID_APPL: Application Code
- \$ID_DATA: Fieldbus specific data
- \$ID_ERR: error Code
- \$ID_IDX: Device index
- \$ID_ISIZE: Input bytes quantity
- \$ID_ISMST: this Device is a Master (or Controller)
- \$ID_MASK: miscellaneous mask of bits
- \$ID_NAME: Device name
- \$ID_NET: Fieldbus type
- \$ID_NOT_ACTIVE: not active Device at net boot time
- \$ID_OSIZE: output bytes quantity
- \$ID_STS: Device status
- \$ID_TYPE: configuration status
- \$IN: IN digital
- \$IPERIOD: Interpolator period
- \$IREG: Integer register - saved
- \$IREG_NS: Integer registers - not saved
- \$IR_ARM: Interference Region Arm
- \$IR_CNFG: Interference Region Configuration
- \$IR_ENBL: Interference Region enabled flag
- \$IR_IN_OUT: Interference Region location flag
- \$IR_JNT_N: Interference Region negative joint
- \$IR_JNT_P: Interference Region positive joint
- \$IR_PBIT: Interference Region port bit
- \$IR_PERMANENT: Interference Region permanent
- \$IR_PIDX: Interference Region port index
- \$IR_POS1: Interference Region position
- \$IR_POS2: Interference Region position
- \$IR_POS3: Interference Region position
- \$IR_POS4: Interference Region position
- \$IR_PTYPE: Interference Region port type
- \$IR_REACHED: Interference Region position reached flag
- \$IR_SHAPE: Interference Region Shape

- \$IR_TBL: Interference Region table data
- \$IR_TYPE: Interference Region type
- \$IR_UFRAME: Interference Region Uframe
- \$IS_DEV: associated Device
- \$IS_HELP_INT: User Help code
- \$IS_HELP_STR: User Help string
- \$IS_NAME: I/O point name
- \$IS_PORT_CODE: Port type
- \$IS_PORT_INDEX: Port type index
- \$IS_PRIV: privileged Port flag
- \$IS_RTN: retentive output flag
- \$IS_SIZE: Port dimension
- \$IS_STS: Port status
- \$JERK: Jerk control values
- \$JL_TABLE: Internal Arm data
- \$JNT_LIMIT_AREA: Joint limits of the work area
- \$JNT_MASK: Joint arm mask
- \$JNT_MTURN: Check joint Multi-turn
- \$JNT_OVR: joint override
- \$JNT_SM: joint trajectory planning indicator
- \$JOG_INCR_DIST: Increment Jog distance
- \$JOG_INCR_ENBL: Jog incremental motion
- \$JOG_INCR_ROT: Rotational jog increment
- \$JOG_SPD_OVR: Jog speed override
- \$JPAD_DIST: Distance between user and Jpad
- \$JPAD_ORNT: Teach Pendant Angle setting
- \$JPAD_TYPE: TP Jpad modality rotational or translational
- \$JREG: Jointpos registers - saved
- \$JREG_NS: Jointpos register - not saved
- \$L_ALONG_1D: Internal Loop data
- \$L_ALONG_2D: Internal Loop data
- \$L_AREAL_1D: Internal Loop data
- \$L_AREAL_2D: Internal Loop data
- \$LATCH_CNG: Latched alarm configuration setting
- \$LIN_ACC_LIM: Linear acceleration limit
- \$LIN_DEC_LIM: Linear deceleration limit
- \$LIN_SPD: Linear speed
- \$LIN_SPD_LIM: Linear speed limit
- \$LIN_SPD_RT: Run-time Linear speed

- \$LIN_SPD_RT_OVR: Run-time Linear speed override
- \$LOG_TO_CHANNEL: Logical to channel relationship
- \$LOG_TO_DRV: Logical to physical drives relationship
- \$LOOP_DATA: Loop data
- \$MAIN_JNTP: PATH node main jointpos destination
- \$MAIN_POS: PATH node main position destination
- \$MAIN_XTND: PATH node main xtndpos destination
- \$MAN_SCALE: Manual scale factor
- \$MDM_INT: Modem Configuration
- \$MDM_STR: Modem Configuration
- \$MOD_ACC_DEC: Modulation of acceleration and deceleration
- \$MOD_MASK: Joint mod mask
- \$MOVE_STATE: Move state
- \$MOVE_TYPE: Type of motion
- \$MTR_ACC_TIME: Motor acceleration time
- \$MTR_CURR: Motor current
- \$MTR_DEC_TIME: Motor deceleration time
- \$MTR_SPD_LIM: Motor speed limit
- \$NET_B: Ethernet Boot Setup
- \$NET_B_DIR: Ethernet Boot Setup Directory
- \$NET_C_CNGF: Ethernet Client Setting Modes
- \$NET_C_DIR: Ethernet Client Setup Default Directory
- \$NET_C_HOST: Ethernet Client Setup Remote Host
- \$NET_C_PASS: Ethernet Client Setup Password
- \$NET_C_USER: Ethernet Client Setup Login Name
- \$NET_HOSTNAME: Ethernet network hostnames
- \$NET_I_INT: Ethernet Network Information (integers)
- \$NET_I_STR: Ethernet Network Information (strings)
- \$NET_L: Ethernet Local Setup
- \$NET_MOUNT: Ethernet network mount
- \$NET_Q_STR: Ethernet Remote Interface Information
- \$NET_R_STR: Ethernet Remote Interface Setup
- \$NET_S_INT: Ethernet Network Server Setup
- \$NET_T_HOST: Ethernet Network Time Protocol Host
- \$NET_T_INT: Ethernet Network Timer
- \$NOLOG_ERROR: Exclude messages from logging
- \$NUM ALOG_FILES: Number of action log files
- \$NUM_ARMS: Number of arms
- \$NUM_AUX_AXES: Number of auxiliary axes

- \$NUM_DBs: Number of allowed Databases
- \$NUM_DEVICES: Number of Devices
- \$NUM_JNT_AXES: Number of joint axes
- \$NUM_LUNS: Number of LUNs
- \$NUM_MB: Number of motion buffers
- \$NUM_MB_AHEAD: Number of motion buffers ahead
- \$NUM_PROGS: Number of active programs
- \$NUM_SCRNS: Number of screens
- \$NUM_TIMERS: Number of timers
- \$NUM_VP2_SCRNS: Number of Visual PDL2 screens
- \$NUM_WEAVES: Number of weaves (WEAVE_TBL)
- \$ODO_METER: average TCP space
- \$ON_POS_TBL: ON POS table data
- \$OP_JNT: On Pos jointpos
- \$OP_JNT_MASK: On Pos Joint Mask
- \$OP_POS: On Pos position
- \$OP_REACHED: On Pos position reached flag
- \$OP_TOL_DIST: On Pos-Jnt Tolerance distance
- \$OP_TOL_ORNT: On Pos-Jnt Tolerance Orientation
- \$OP_TOOL: The On Pos Tool
- \$OP_TOOL_DSBL: On Pos tool disable flag
- \$OP_TOOL_RMT: On Pos Remote tool flag
- \$OP_UFRAME: The On Pos Uframe
- \$ORNT_TYPE: Type of orientation
- \$OT_COARSE: On Trajectory indicator
- \$OT_JNT: On Trajectory joint position
- \$OT_POS: On Trajectory position
- \$OT_TOL_DIST: On Trajectory Tolerance distance
- \$OT_TOL_ORNT: On Trajectory Orientation
- \$OT_TOOL: On Trajectory TOOL position
- \$OT_TOOL_RMT: On Trajectory remote tool flag
- \$OT_UFRAME: On Trajectory User frame
- \$OT_UNINIT: On Trajectory position uninit flag
- \$OUT: OUT digital
- \$PAR: Nodal motion variable
- \$PGOV_ACCURACY: required accuracy in cartesian motions
- \$PGOV_MAX_SPD_REDUCTION: Maximum speed scale factor
- \$PGOV_ORNT_PER: percentage of orientation
- \$POS_LIMIT_AREA: Cartesian limits of work area

- \$PREG: Position registers - saved
- \$PREG_NS: Position registers - not saved
- \$PROG_ACC_OVR: Program acceleration override
- \$PROG_ARG: Program's activation argument
- \$PROG_ARM: Arm of program
- \$PROG_ARM_DATA: Arm-related Data - Program specific
- \$PROG_CNFG: Program configuration
- \$PROG_CONDS: Defined conditions of a program
- \$PROG_DEC_OVR: Program deceleration override
- \$PROG_LINE: executing program line
- \$PROG_LUN: program specific default LUN
- \$PROG_NAME: Executing program name
- \$PROG_OWNER: Program Owner of executing line
- \$PROG_RES: Program's execution result
- \$PROG_RREG: Real Registers - Program specific
- \$PROG_SPD_OVR: Program speed override
- \$PROP_AUTHOR: last Author who saved the file
- \$PROP_DATE: date and time when the program was last saved
- \$PROP_FILE: property information for loaded file
- \$PROP_HELP: the help the user wants
- \$PROP_HOST: Controller ID or PC user domain, upon which the file was last saved
- \$PROP_REVISION: user defined string representing the version
- \$PROP_TITLE: the title defined by the user for the file
- \$PROP_UML: user modify level
- \$PROP_UVL: user view level
- \$PROP_VERSION: version upon which it was built
- \$PWR_RCVR: Power failure recovery mode
- \$RAD_IDL_QUO: Radian ideal quote
- \$RAD_TARG: Radian target
- \$RAD_VEL: Radian velocity
- \$RBT_CNFG: Robot board configuration
- \$RB_FAMILY: Family of the robot arm
- \$RB_MODEL: Model of the robot arm
- \$RB_NAME: Name of the robot arm
- \$RB_STATE: State of the robot arm
- \$RB_VARIANT: Variant of the robot arm
- \$RCVR_DIST: Distance from the recovery position
- \$RCVR_LOCK: Change arm state after recovery

- \$RCVR_TYPE: Type of motion recovery
- \$READ_TOUT: Timeout on a READ
- \$REC_SETUP: RECord key setup
- \$REMOTE: Functionality of the key in remote
- \$REM_I_STR: Remote connections Information
- \$REM_TUNE: Internal remote connection tuning parameters
- \$RESTART: Restart Program
- \$RESTART_MODE: Restart mode
- \$RESTORE_SET: Default devices
- \$ROT_ACC_LIM: Rotational acceleration limit
- \$ROT_DEC_LIM: Rotational deceleration limit
- \$ROT_SPD: Rotational speed
- \$ROT_SPD_LIM: Rotational speed limit
- \$RREG: Real registers - saved
- \$RREG_NS: Real registers - not saved
- \$SAFE_ENBL: Safe speed enabled
- \$SAFE_SPD: User Velocity for testing safety speed
- \$SAFE_SPD: User Velocity for testing safety speed
- \$SDO: System digital output
- \$SEG_ADV: PATH segment advance
- \$SEG_COND: PATH segment condition
- \$SEG_DATA: PATH segment data
- \$SEG_FLY: PATH segment fly or not
- \$SEG_FLY_DIST: Parameter in segment fly motion
- \$SEG_FLY_PER: PATH segment fly percentage
- \$SEG_FLY_TRAJ: Type of fly control
- \$SEG_FLY_TYPE: PATH segment fly type
- \$SEG_OVR: PATH segment override
- \$SEG_REF_IDX: PATH segment reference index
- \$SEG_STRESS_PER: Percentage of stress required in fly
- \$SEG_TERM_TYPE: PATH segment termination type
- \$SEG_TOL: PATH segment tolerance
- \$SEG_TOOL_IDX: PATH segment tool index
- \$SEG_WAIT: PATH segment WAIT
- \$SENSOR_CNVRSN: Sensor Conversion Factors
- \$SENSOR_ENBL: Sensor Enable
- \$SENSOR_GAIN: Sensor Gains
- \$SENSOR_OFST_LIM: Sensor Offset Limits
- \$SENSOR_TIME: Sensor Time

- \$SENSOR_TYPE: Sensor Type
- \$SERIAL_NUM: Serial Number of the board
- \$SFRAME: Sensor frame of an arm
- \$SING_CARE: Singularity care
- \$SM4C_SAT_VEL_SCALE: thresholds for the joint speed saturation in Cartesian SmartMove
- \$SM4C_STRESS_PER: Maximum Stress allowed in Cartesian SmartMove
- \$SM4_SAT_ADD_SCALE: Additional threshold for the SmartMove saturation
- \$SPD_OPT: Type of speed control
- \$SREG: String registers - saved
- \$SREG_NS: String registers - not saved
- \$STARTUP: Startup program
- \$STARTUP_USER: Configuration file name
- \$STRESS_PER: Stress percentage in cartesian fly
- \$STRK_END_N: User negative stroke end
- \$STRK_END_P: User positive stroke end
- \$STRK_END_SYS_N: System stroke ends
- \$STRK_END_SYS_P: System stroke ends
- \$SYNC_ARM: Synchronized arm of program
- \$SYS_CALL_OUT: Output lun for SYS_CALL
- \$SYS_CALL_STS: Status of last SYS_CALL
- \$SYS_CALL_TOUT: Timeout for SYS_CALL
- \$SYS_ERROR: Last system error
- \$SYS_ID: Robot System identifier
- \$SYS_OPTIONS: System options
- \$SYS_PARAMS: Robot system identifier
- \$SYS_RESTART: System Restart Program
- \$SYS_STATE: State of the system
- \$TERM_TYPE: Type of motion termination
- \$THRD_CEXP: Thread Condition Expression
- \$THRD_ERROR: Error of each thread of execution
- \$THRD_PARAM: Thread Parameter
- \$TIMER: Clock timer (in ms)
- \$TOL_ABT: Tolerance anti-bounce time
- \$TOL_COARSE: Tolerance coarse
- \$TOL_FINE: Tolerance fine
- \$TOL_JNT_COARSE: Tolerance for joints
- \$TOL_JNT_FINE: Tolerance for joints
- \$TOL_TOUT: Tolerance timeout

- \$TOOL: Tool of arm
- \$TOOL_CNTR: Tool center of mass of the tool
- \$TOOL_INERTIA: Tool Inertia
- \$TOOL_MASS: Mass of the tool
- \$TOOL_RMT: Fixed Tool
- \$TOOL_XTREME: Extreme Tool of the Arm
- \$TP_ARM: Teach Pendant current arm
- \$TP_GEN_INCR: Incremental value for general override
- \$TP_MJOG: Type of TP jog motion
- \$TP_ORNT: Orientation for jog motion
- \$TP_SYNC_ARM: Teach Pendant's synchronized arms
- \$TRK_TBL: Tracking application table data
- \$TT_APPL_ID: Tracking application identifier
- \$TT_ARM_MASK: Tracking arm mask
- \$TT_FRAMES: Array of POSITIONS for tracking application
- \$TT_I_PRMS: Integer parameters for the tracking application
- \$TT_PORT_IDX: Port Index for the tracking application
- \$TT_PORT_TYPE: Port Type for the tracking application
- \$TT_R_PRMS: Real parameters for the tracking application
- \$TUNE: Internal tuning parameters
- \$TURN_CARE: Turn care
- \$TX_RATE: Transmission rate
- \$UDB_FILE: Name of UDB file
- \$UFRAME: User frame of an arm
- \$VERSION: Software version
- \$VP2_SCRN_ID: Executing program VP2 Screen Identifier
- \$VP2_TOUT: Timeout value for asynchronous VP2 requests
- \$VP2_TUNE: Visual PDL2 tuning parameters
- \$WEAVE_MODALITY: Weave modality
- \$WEAVE_MODALITY_NOMOT: Weave modality (only for no arm motion)
- \$WEAVE_NUM: Weave table number
- \$WEAVE_NUM_NOMOT: Weave table number (only for no arm motion)
- \$WEAVE_PHASE: Index of the Weaving Phase
- \$WEAVE_TBL: Weave table data
- \$WEAVE_TYPE: Weave type
- \$WEAVE_TYPE_NOMOT: Weave type (only for no arm motion)
- \$WFR_IOTOUT: Timeout on a WAIT FOR when IO simulated
- \$WFR_TOUT: Timeout on a WAIT FOR
- \$WORD: WORD data

- \$WRITE_TOUT: Timeout on a WRITE
- \$WV_AMPLITUDE: Weave amplitude
- \$WV_AMP_PER_RIGHT/LEFT: Weave amplitude percentage
- \$WV_CNTR_DWL: Weave center dwell
- \$WV_DIRECTION_ANGLE: Weave angle direction
- \$WV_END_DWL: Weave end dwell
- \$WV_LEFT_AMP: Weave left amplitude
- \$WV_LEFT_DWL: Weave left dwell
- \$WV_LENGTH_WAVE: Wave length
- \$WV_ONE_CYCLE: Weave one cycle
- \$WV_PLANE: Weave plane angle
- \$WV_PLANE_MODALITY: Weave plane modality
- \$WV_RADIUS: Weave radius
- \$WV_RIGHT_AMP: Weave right amplitude
- \$WV_RIGHT_DWL: Weave right dwell
- \$WV_SMOOTH: Weave smooth enabled
- \$WV_SPD_PROFILE: Weave speed profile enabled
- \$WV_TRV_SPD: Weave transverse speed
- \$WV_TRV_SPD_PHASE: Weave transverse speed phase
- \$XREG: Xtndpos registers - saved
- \$XREG_NS: Xtndpos registers - not saved

12.10 \$A_ALONG_1D: Internal arm data

Memory category: field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: There are certain fields of \$ARM_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields are reserved.

12.11 \$A_ALONG_2D: Internal arm data

Memory category: field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of integer of two dimension

Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$ARM_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields are reserved.

12.12 \$A_AREAL_1D: Internal arm data

Memory category field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$ARM_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields are reserved.

12.13 \$A_AREAL_2D: Internal arm data

Memory category field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of real of two dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$ARM_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields are reserved.

12.14 \$AIN: Analog input

Memory category port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits none
S/W Version: 1.0

Default:

Description: It represents the analog input points. For further information see also [par. 5.2.3 \\$AIN and \\$AOUT on page 104](#).

12.15 \$AOUT: Analog output

<i>Memory category</i>	port
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the analog output points. For further information see also par. 5.2.3 \$AIN and \$AOUT on page 104 .

12.16 \$APPL_ID: Application Identifier

<i>Memory category</i>	static
<i>Load category:</i>	retentive
<i>Data type:</i>	string
<i>Attributes:</i>	property
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable contains the application identifier, similar to the system identifier, except that it can be configured by the user / application. This field should be configured to be unique per controller.

12.17 \$APPL_NAME: Application Identifiers

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category - environment</i>
<i>Data type:</i>	array of string of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	Each element of this array is a string that contains the name of an application that is running on the Controller

12.18 \$APPL_OPTIONS: Application Options

<i>Memory category</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable contains the definitions of the application features currently enabled on the Controller

12.19 \$ARM_ACC_OVR: Arm acceleration override

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - overrides</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	property; WITH MOVE; MOVE ALONG
<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the acceleration override percentage for motions issued to a specific arm. There is one value per arm. Maximum speed and deceleration are not affected by changes in its value. Changes in \$ARM_ACC_OVR take effect on the next motion and for the entire motion.

12.20 \$ARM_DATA: Robot arm data

<i>Memory category</i>	dynamic
<i>Load category:</i>	not saved
<i>Data type:</i>	array of arm_data of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	\$ARM_DATA is an array of predefined records with one element for each arm. The fields of each element represent arm-related data. It is not always necessary to specify the \$ARM_DATA prefix when referring to a field of \$ARM_DATA. The field of \$ARM_DATA that is used by default is the one for the arm specified in PROG_ARM.

```

PROGRAM armdata PROG_ARM=1
VAR init_tool : POSITION
BEGIN
  $ARM_DATA[2].TOOL := init_tool -- arm 2
  $TOOL := init_tool -- arm 1
END armdata
  
```

12.21 \$ARM_DEC_OVR: Arm deceleration override

<i>Memory category:</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - overrides</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	WITH MOVE; MOVE ALONG
<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the deceleration override percentage for motions issued to a particular arm. There is one value per arm. Acceleration and maximum speed are not affected by changes in its value. Changes in \$ARM_DEC_OVR take effect on the next motion and for the entire motion.

12.22 \$ARM_DISB: Arm disable flags

<i>Memory category:</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the current state of the arm and whether it is enabled for motion and with the DRIVEs ON, or it is disabled. Issuing a DRIVE ON in PROG when the robot is in the disabled state will cause a warning message to be displayed. A value of TRUE means that the arm is disabled

12.23 \$ARM_ID: Arm identifier

<i>Memory category:</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - configuration</i>
<i>Data type:</i>	string
<i>Attributes:</i>	privileged read-write, property
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0

Description: This field of \$ARM_DATA contains the arm identifier. It is used during the System installation phase.

12.24 \$ARM_LINKED: Enable/disable arm coupling

Memory category: field of arm_data

Load category: arm *Minor category* - configuration

Data type: boolean

Attributes: WITH MOVE

Limits: ON..OFF

Default: OFF

S/W Version: 1.0

Description: It is used in configurations in which an arm is mechanically linked to another (integrated arms). In this case the first arm (the one directly linked to the world frame) can move in a coupled way (\$ARM_DATA[previuos_arm].ARM_LINKED=TRUE) or not (\$ARM_DATA[previous_arm].ARM_LINKED=FALSE). If the first arm moves in a coupled way, the next arm (linked to the flange of the first one) moves as well to maintain its TCP steady. On the contrary, if \$ARM_LINKED is set to the FALSE value, the next arm will be simply carried by the previous maintaining its axes steady (as a consequence its TCP will move). No move for the next arm can start during a non coupled movement of the previous.

The variable is meaningful only for the first of an integrated couple of arms; it has no effect in other situations.

12.25 \$ARM_OVR: Arm override

Memory category: field of arm_data

Load category: arm. *Minor category* - overrides

Data type: integer

Attributes: property

Limits: 1..100

Default: 100

S/W Version: 1.0

Description: \$ARM_OVR is similar to \$GEN_OVR but is accessible from a PDL2 program, whereas \$GEN_OVR is accessible from the Teach Pendant. The variable scales speed, acceleration and deceleration, so that the trajectory remains constant with changes in its value. A change to \$ARM_OVR immediately effects the shape of the velocity profile.

12.26 \$ARM_SENSITIVITY: Arm collision sensitivity

Memory category: field of arm_data

Load category: arm. *Minor category* - collision

Data type: array of integer of two dimension

Attributes: WITH MOVE
Limits 0..100
Default:
S/W Version: 1.0
Description: This array represents the sensitivity threshold on each robot axis in the collision detection algorithm.
 The first dimension is the one of the possible values of \$COLL_TYPE. The second dimension is the axis number.
 The value of each element must stay in the range 0..100.

12.27 \$ARM_SIMU: Arm simulate flag

Memory category field of crnt_data
Load category: not saved
Data type: boolean
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: It represents the current state of the arm and whether it is in simulate mode or not. In this mode, motion can be interpreted on the arm, but there is no physical motion. This is useful for testing the timing and flow of a program without causing any motion on the arm. A value of TRUE means that the arm is in the simulated state.

12.28 \$ARM_SPACE: current Arm Space

Memory category field of crnt_data
Load category: not saved
Data type: real
Attributes: PDL2 read-only
Limits 1..100
Default:
S/W Version: 1.0
Description: It represents the current arm space. It is only used in cartesian motions for indicating the average TCP space expressed in millimeters.
 This variable is updated every tick (default 2 milliseconds) in case of linear or circular motions and every 5 ticks in case of joint movements.

```

PROGRAM curspace NOHOLD
BEGIN
CYCLE
  WRITE ($CRNT_DATA [1] .ARM_SPACE, NL)
  DELAY 500
END curspace
  
```

12.29 \$ARM_SPD_OVR: Arm speed override

Memory category: field of arm_data
Load category: arm. *Minor category* - overrides
Data type: integer
Attributes: property; WITH MOVE; MOVE ALONG
Limits: 1..100
Default: 100
S/W Version: 1.0
Description: It represents the speed override percentage for motions issued to a specified arm. There is one value per arm. Acceleration and deceleration are not affected by changes in its value. Changes in \$ARM_SPD_OVR take effect on the next motion and for the entire motion.

12.30 \$ARM_USED: Program use of arms

Memory category: static
Load category: not saved
Data type: array of boolean of one dimension
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: Each element of this array of booleans indicates whether at least one program is active on that arm. The array index corresponds to the arm number. If \$ARM_USED[1] is TRUE, it means that on arm 1 there is one holdable program running.

12.31 \$ARM_VEL: Arm velocity

Memory category: field of crnt_data
Load category: not saved
Data type: real
Attributes: read-only, property
Limits: none
Default:
S/W Version: 1.0
Description: It represents the current arm velocity. It is meaningful only in cartesian motions for indicating the average TCP velocity expressed in meters per second. This variable is updated every tick (default 2 milliseconds) in case of linear or circular motions and every 100 ms in case of joint movements.

```
PROGRAM vel NOHOLD
BEGIN CYCLE
    WRITE ($CRNT_DATA [1].ARM_VEL, NL)
    DELAY 500
END vel
```

12.32 \$AUX_BASE: Auxiliary base for a positioner of an arm

Memory category: field of arm_data
Load category: arm. *Minor category - auxiliary*
Data type: array of position of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the location and orientation of the base of a positioner with respect to the world frame. It is used for positioners integrated in the same arm of the robot by means of auxiliary axes. There is an array element for each possible positioner.

12.33 \$AUX_KEY: Teach Pendant AUX-A and AUX-B keys mapping

Memory category: field of arm_data
Load category: arm. *Minor category - auxiliary*
Data type: array of integer of one dimension
Attributes: none
Limits: 7..10
Default: 7
S/W Version: 1.0
Description: This variable contains the indication of the auxiliary axis that is associated to the corresponding key on the Teach Pendant. Element 1 is related to the AUX-A key and element 2 is related to AUX-B key.

12.34 \$AUX_MASK: Auxiliary arm mask

Memory category: field of arm_data
Load category: arm. *Minor category - auxiliary*
Data type: integer
Attributes: privileged read-write, property
Limits: none
Default:
S/W Version: 1.0
Description: Note htat this mask only refers to the auxiliary axes.
 Each bit in this INTEGER data represents whether the corresponding auxiliary axis is present for the arm or not.
 Example: if the current arm just includes axis 7, the corresponding \$AUX_MASK value must be 0x40.

12.35 \$AUX_OFST: Auxiliary axes offsets

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - auxiliary
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	WITH MOVE; MOVE ALONG
<i>Limits</i>	
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable allows to define an offset for each auxiliary axis. There is an array element for each auxiliary axis. These offsets are added to the corresponding fields of the extended positions (XTNDPOS) while they are executed in a motion statement. This is mainly useful with integrated axes, like rails or rotating columns on which the robot is mounted, to execute a program in different positions in respect to the original one. In these cases infact, \$UFRAME is used to translate the cartesian component of the XTNDPOS while \$AUX_OFST acts on the auxiliary axes components. \$AUX_OFST is only used on auxiliary axes defined as positioners, integrated axes or electrical welding guns. The DisplayArmPosition command takes into account these offsets.

12.36 \$AUX_TYPE: Positioner type

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It reports a description of the positioners enabled for the cooperative motion. The first dimension is an INTEGER used to set the type of positioner and the second is a bit mask of the axes involved in the positioner. Any change to the value of this variable immediately takes effect

12.37 \$AX_CNVRSN: Axes conversion

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0

Description: It represents the conversion factor between radians and degrees for each rotational axis and between millimeters and millimeters for each linear axis. There is an array element for each axis.

```
PROGRAM quo NOHOLD
VAR i : INTEGER
  gr_quo : ARRAY [6] OF REAL
BEGIN
  FOR i := 1 TO 6 DO
    -- Conversion of the quote
    -- measured with axes
    -- radiant in axes degrees
    gr_quo[i] :=
      $CRNT_DATA[1].RAD_IDL_QUO[i] *
      $ARM_DATA[1].AX_CNVRSN[i]
  ENDFOR
END quo
```

12.38 \$AX_INF: Axes inference

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: array of real of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It represents the inference between two axes, expressed in radians, when a rotation of one axis affects another.

```
PROGRAM quo NOHOLD
VAR i : INTEGER
  gr_quo : ARRAY [6] OF REAL
BEGIN
  FOR i := 1 TO 6 DO
    -- Transform between the quota
    -- expressed in radians
    -- and the angle of the axes
    gr_quo[i] := 0
  IF i = 6 THEN
    gr_quo[6] := $CRNT_DATA[1].RAD_IDL_QUO[5] *
      $ARM_DATA[1].AX_INF[6] *
      $ARM_DATA[1].AX_CNVRSN[6]
  ENDIF
  gr_quo[i] := gr_quo[i] +
    $CRNT_DATA[1].RAD_IDL_QUO[i] *
    $ARM_DATA[1].AX_CNVRSN[i]
  ENDFOR
END quo
```

12.39 \$AX_LEN: Axes lengths

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: it represents the axes lengths of the arm measured in millimeters.

12.40 \$AX_OFST: Axes offsets

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the offset between two consecutive axes. For example, some SMART machines have an offset between the base and the pivot of the second axis.

12.41 \$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature

Memory category: arm_data
Load category: controller. *Minor category - environment*
Data type: integer
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is the Arm which acts as Master in the Axes Pursuit feature
See also [\\$AX_PURSUIT_ENBL: Axes Pursuit enabling flag](#),
 [\\$AX_PURSUIT_LINKED: Axes Pursuit linked](#).

12.42 \$AX_PURSUIT_ENBL: Axes Pursuit enabling flag

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	property
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable is used for enabling (TRUE) and disabling (FALSE) the Axes Pursuit feature. Please note that if the user sets this variable (TRUE), without purchasing the feature, the system automatically sets it FALSE again.
<i>See also</i>	\$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature , \$AX_PURSUIT_LINKED: Axes Pursuit linked .

12.43 \$AX_PURSUIT_LINKED: Axes Pursuit linked

<i>Memory category:</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - configuration</i>
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	The value of each array element identifies which axis of the Master arm (\$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature) is followed by the axis (of the arm indicated by \$ARM_DATA index) corresponding to the array index with a positive/negative direction.

For example:

```
$AX_PURSUIT_ARM_MASTER := 1
$ARM_DATA[2].AX_PURSUIT_LINKED[3] := 7
```

means that axis 3 of Arm 2 (SLAVE arm) pursuits axis 7 of Arm 1 (MASTER Arm).

For example:

```
$AX_PURSUIT_ARM_MASTER := 1
$ARM_DATA[2].AX_PURSUIT_LINKED[3] := -7
```

means that axis 3 of Arm 2 (SLAVE arm) pursuits axis 7 of Arm 1 in the negative direction (MASTER Arm).

<i>See also</i>	\$AX_PURSUIT_ARM_MASTER: Master Arm in the Axes Pursuit feature , \$AX_PURSUIT_ENBL: Axes Pursuit enabling flag .
-----------------	--

12.44 \$B_ALONG_1D_NS: Internal board data

Memory category: static
Load category: not saved
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: This predefined variable is not saved and can be referenced as a general array. The format and meaning of the data within these fields is reserved.

12.45 \$B_ASTR_1D_NS: Board string data

Memory category: field of arm_data
Load category: not saved
Data type: array of string of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: An array that contains string board data that is not saved.
 Element
 1 : the BSP version
 2 : the build version
 3 : the TP version
 4 : Mac identifier for fei0
 5 : Mac identifier for fei1

12.46 \$BACKUP_SET: Default devices

Memory category: static
Load category: controller *Minor category* - environment
Data type: array of string of one dimension
Attributes: none
Limits: none
Default: The default values are as follows:
 – \$BACKUP_SET[1] - 'All|UD:*.*/s'
 – \$BACKUP_SET[2] - 'SYS|UD:\sys/s/x*.lba/x*.lbe'
 – \$BACKUP_SET[7] - 'Today|UD:*.*/s/a0'
 – \$BACKUP_SET[8] - 'Inc|UD:*.*/s/l'
S/W Version: 1.0

Description: Array of 8 elements with each element representing the definition of a backup Saveset.
 The format is as follows:
`<id>|<filespec> [/<opt>] [/x<filespec>]`
 where '`id`' is the Saveset name, '`<opt>`' includes options and '`/x<filespec>`' defines which files are to be excluded from the backup.
 These Savesets are used by the FilerUtilityBackup command, when the option /S is specified, for understanding where the files should be copied from.
 If, for example, \$BACKUP_SET[1] is set to 'set1|UD:\dir1*.*\s', the command Filer Utility Backup/Saveset issued with the set1 parameter for the save set, will copy all the files found in UD:\dir1, including the subdirectories.
 The following switches are available to properly setup the Saveset:

- **/A** (=After) backup files created/modified BEFORE or AFTER the specified date - an INTEGER indicates the total amount of days BEFORE today; the format is DATE with sign: a negative sign means BEFORE, whereas a positive sign means AFTER the specified date
- **/U** (=Unset) do not set 'a' attribute for a certain file
- **/I** (=Incremental) only backup files which haven't been saved before (i.e. files without 'a' attribute)
- **/S** (=Subdirectories) include subdirectories
- **/X** (=eXclude) exclude certain files from the backup operation

 The following predefined Savesets are available:

- \$BACKUP_SET[1] : "All|TTS_USER_DEV"*.*\s"
- \$BACKUP_SET[7] : "Today|TTS_USER_DEV"*.*\s/a0"
- \$BACKUP_SET[8] : "Inc|TTS_USER_DEV"*.*\s/i"

12.47 \$BASE: Base of arm

Memory category: field of arm_data
Load category: arm. *Minor category* - chain
Data type: position
Attributes: WITH MOVE; MOVE ALONG
Limits: none
Default:
S/W Version: 1.0
Description: It represents the location and orientation of the base of the robot relative to world frame of reference.

12.48 \$BIT: BIT data

Memory category: port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: pulse usable
Limits: none
Default:
S/W Version: 1.0

Description: This structure is a boolean port array; the size of each element is 1 bit.
For further information see also [par. 5.4.1 \\$BIT on page 119](#).

12.49 \$BOOTLINES: Bootline read-only

Memory category static
Load category: retentive
Data type: array of string of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: The bootlines define how the board is to be booted. There are different bootline options and configurations, but these should not be changed by the user.

12.50 \$BREG: Boolean registers - saved

Memory category static
Load category: controller. *Minor category* - vars
Data type: array of boolean of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Boolean registers are global variables that can be used by users instead of having to define variables.
Also for Integer, Real, String, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.51 \$BREG_NS: Boolean registers - not saved

Memory category static
Load category: not saved
Data type: array of boolean of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Non-saved boolean registers that can be used by users instead of having to define variables.
Also for Integer, Real, String, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.52 \$B_NVRAM: NVRAM data of the board

Memory category static
Load category: not saved
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This predefined variable is saved into NVRAM and can be referenced as an array. The format and meaning of the stored data in it is reserved.

12.53 \$C_ALONG_1D: Internal current arm data

Memory category field of crnt_data
Load category: not saved
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$CRNT_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields is reserved

12.54 \$C_AREAL_1D: Internal current arm data

Memory category field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$CRNT_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields is reserved

12.55 \$C_AREAL_2D: Internal current arm data

Memory category field of crnt_data

Load category: not saved
Data type: array of real of two dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: There are certain fields of \$CRNT_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields is reserved

12.56 \$CAL_DATA: Calibration data

Memory category field of arm_data
Load category: arm. *Minor category* - calibration
Data type: arrays of real of one dimension
Attributes: none
Limits -0.5 .. 0.5
Default: 0.0
S/W Version: 1.0
Description: It represents the calibration error difference between the theoretical arm and the actual arm position. The value is expressed in motor turns.

12.57 \$CAL_SYS: System calibration position

Memory category field of arm_data
Load category: arm *Minor category* - calibration
Data type: jointpos
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the system calibration position. For each new machine the user must jog the robot to the calibration position, based on calipers or stamps, and then follow the calibration procedure before using the robot in AUTO mode.

```

PROGRAM calpos PROG_ARM=4
VAR pnt0001j : JOINTPOS
BEGIN
    WRITE($ARM_DATA[4].CAL_SYS, NL)
    pnt0001j := $CAL_SYS
    MOVE JOINT TO $CAL_SYS
END calpos
    
```

12.58 \$CAL_USER: User calibration position

Memory category: field of arm_data
Load category: arm. *Minor category* - calibration
Data type: jointpos
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents a user-defined calibration position. Using this variable, the robot can be calibrated in a position other than the system defined calibration position.

```

PROGRAM userpos PROG_ARM=2
BEGIN
    -- Write the user calibration position and
    -- move to that position
    $CAL_USER := $CAL_SYS
    $CAL_USER[3] := $CAL_USER[3] - 90
    WRITE($ARM_DATA[2].CAL_USER, NL)
    MOVE JOINT TO $CAL_USER
END userpos

```

12.59 \$CAUX_POS: Cartesian positioner position

Memory category: field of arm_data
Load category: not saved
Data type: position
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It contains the cartesian position of the enabled positioner for the cooperative motion.

12.60 \$CIO_TREEID: Tree identifier for the configure I/O tree

Memory category: static
Load category: not saved
Data type: integer
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It contains the value of the Configure I/O Tree Identifier.

12.61 \$CNFG_CARE: Configuration care

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: boolean
Attributes: field node, WITH MOVE; MOVE ALONG
Limits: none
Default: ON
S/W Version: 1.0
Description: Represents a flag to determine whether a Cartesian trajectory with a different starting and ending configuration should be executed. If the flag is ON and the initial and final configurations are different, the motion is not executed. If the flag is OFF, the final configuration will be the same as the initial configuration without any messages.

12.62 \$CNTRL_CNFG: Controller configuration mode

Memory category: static
Load category: controller. *Minor category* - environment
Data type: integer
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Different bits of this predefined variable are used for setting the configuration and startup of the controller:

- Bit 1: If set, the pop-up window containing informations on the system configuration is not displayed at system restart.
- Bit 2: Disables the checks performed at I/O setting, on: TP position (on/off cabinet), TP connection status.
This means that , if this bit is set to 1, bit 3 is not considered.
- Bit 3: If set, all the programs activated in the system will have bit 3 in \$PROG_CNFG set, which disables the setting of an output in these circumstances:
 - a: if in PROGR state from an active program.
 - b: if in PROGR state executing the statement from the WINC5G program running on the PC when the Teach Pendant is recognized to be out of cabinet.
 - c: if in PROGR state under MEMORY DEBUG or PROGRAM EDIT from the WINC5G program running on the PC when the Teach Pendant is recognized to be out of cabinet.
 - d: if the state selector was turned out of the T1 position when the Teach Pendant is recognized to be out of cabinet.
- Bit 4: reserved
- Bit 5: If set, the WAIT FOR does not trigger if the program is in the held state (due to HOLD or to an error)
- Bit 6: reserved
- Bit 7: If set, the communication protocol for WINC5G program is automatically

- mounted on COMP: port after a restart.
- Bit 8..16: reserved
- Bit 17: do not log download messages
- Bit 18: do not log upload messages
- Bit 19-22: reserved
- Bit 23: set to 1 to enable Autosave in IDE and DATA when switching to AUTO state.
- Bit 24: set to 1 if the system is configured to handle the Customized Nodal Move
- Bit 25: Manage UNICODE in the string \$PROG_CNFG set to this
- Bit 26: set to 1 to view the name in the shortest format (e.g. 8 characters in ProgramView for the ProgramName) in some of the views. Otherwise view in the extended format.
- Bit 27: Disables the commands for adding (CCLA) and deleting (CCLD) a login from TP-INT, SYS_CALL and TP Pages.
- Bit 28: set to 1 to disable the property output to the LSV file. The default is that the information is added at the head of the LSV file. Setting this bit to 1 will disable saving of Property information. This can be useful for maintaining upwards compatibility in the LSV file.
- Bit 29: set to 1 to use greater precision when printing REAL number in LSV
- Bit 30-32: reserved

12.63 \$CNTRL_DST: Controller Day Light Saving

Memory category: static

Load category: controller. *Minor category* - environment

Data type: string

Attributes: none

Limits: none

Default: European "6,1,3,1,6,1,10,2"

S/W Version: 1.0

Description: Defines the Day Light Saving settings for the controller. The data can be in one of two formats. Either
<from period> <to_period> with a period defined as First|Second|Third|Fourth|Fifth|Last, Day of week, Month and time. For example
for Europe (which is the default) it is Last Sunday of March at 1:00am to Last Sunday of October, at 1:00am
is "6,1,3,1,6,1,10,1"; or mmddhh:mmddhh

12.64 \$CNTRL_INIT: Controller initialization mode

Memory category: static

Load category: controller. *Minor category* - environment

Data type: integer

Attributes: privileged read-write

Limits: none

Default:

S/W Version: 1.0

- Description:* Each bit represents how different aspects of the controller behave upon initialization:
- Bit 1..6: reserved
 - Bit 7: If set, remote signals are ignored in AUTO and PROGR state.
 - Bit 8..10: reserved
 - Bit 11: This bit configures the remote output as a copy of the state selector in REMOTE position.
 - Bit 12: reserved
 - Bit 13: This bit configures the enabling device switch as the DRIVE OFF button, when released in PROGR instead of HOLD.
 - Bit 14: This bit enables warnings generated when a command is received from a REMOTE device (Remote CAN I/O, FieldBus)
 - Bit 15..16: reserved
 - Bit 17: Enables software timer (1.4 sec) that activate motors brake after remote DRIVE OFF command (used as CONTROLLED EMERGENCY).
 - Bit 18: If set, it disables multiarm command forcing.
 - Bit 19, 20: reserved
 - Bit 21: If set, it means the loading system software is running.
 - Bit 22: If set, ConfigureArmCalibrate and ConfigureArmTurnset commands are disabled.
 - Bit 23: If set, the precision (via the ‘.’ operator), in small orientation variations, around X and Y axes, is of 0.2 degrees (instead of the default of 0.02 degrees). The precision obtained is lower in this case.

12.65 \$CNTRL_OPTIONS: Controller Options

- Memory category*: static
- Load category*: controller
- Data type*: array of integer of one dimension
- Attributes*: read-only
- Limits*: none
- Default*:
- S/W Version*: 1.0
- Description*: Each bit of this variable corresponds to a software feature of the Controller which could be available (bit set to 1) or not (bit set to 0). Here below is described the meaning of each bit of element [1]:
- Bit 1 for Synchronized arm motion
 - Bit 2 for Cooperative motion
 - Bit 3 for Sensor Tracking motion
 - Bit 4 for Conveyor Tracking motion
 - Bit 5 for Weaving motion
 - Bit 6 for Robot Absolute Accuracy
 - Bit 7 for Collision detection
 - Bit 8 for Automatic payload identification algorithm
 - Bit 9 for joint Soft Servo
 - Bit 10..12 reserved
 - Bit 13 for PDL2 read/write on TCP/IP
 - Bit 14 Advanced interference regions (Hybrid)
 - Bit 15 Low resolution Euler angles
 - Bit 16 Speed Control for Arm
 - Bit 17 Axes Pursuit

- Bit 18 reserved
- Bit 19 for SmartMove
- Bit 20 for Path Governor
- Bit 21 reserved
- Bit 22 for VP2.Frames
- Bit 23 for Vp2.Builder
- Bit 24 reserved
- Bit 25 for Multipass
- Bit 26..28 reserved
- Bit 29 for Manual Guidance
- Bit 30 for Palletizing Motion
- Bit 31 Interference regions (monitored and forbidden).

12.66 \$CNTRL_TZ: Controller Time Zone

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category - environment</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It is the difference in time between the Controller location time and Greenwich Mean time.</p> <p>It is needed for the handling of e-mail protection. Note that the Day Light Saving time also needs to be taken into consideration.</p>

12.67 \$COLL_EFFECT: Collision Effect on the arm status

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - configuration</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	MNL_COLL_DRIVEOFF MNL_COLL_EVENT
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It is possible to define the effect of the Collision Alarm (62513 - Collision detected) on the Arm. This predefined variable can assume the following values:</p> <ul style="list-style-type: none"> – 0: a DRIVE OFF is issued. This is the default value. – 1: a HOLD is generated – 2: the collision causes an event which can be tested using WHEN EVENT 197.

12.68 \$COLL_ENBL: Collision enabling flag

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	property
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	When set to TRUE, this flag enables the "Collision detection" feature and has an immediate effect on any movements. The variable is automatically set to FALSE upon DRIVE OFF. It is therefore necessary, after a DRIVE OFF, to re-enable the Collision Detection functionality by assigning to this variable the value of TRUE.

12.69 \$COLL_SOFT_PER: Collision compliance percentage

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - collision
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	WITH MOVE
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents the compliance percentage of each robot axis in the Soft Servo condition caused by a collision. The maximum compliance is 100 and the minimum compliance is 0 in the range defined in \$MOD_ACC_DEC[124] and \$MOD_ACC_DEC[125] for axes 1, 2 and 3, and by values \$MOD_ACC_DEC[126] and \$MOD_ACC_DEC[127] for axes 4, 5 and 6.</p> <p>The first dimension is the one of the possible values of \$COLL_TYPE. The second one is the axis number.</p>

12.70 \$COLL_TYPE: Type of collision

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category</i> - collision
<i>Data type:</i>	integer
<i>Attributes:</i>	property, WITH MOVE; MOVE ALONG
<i>Limits</i>	COLL_DISABLE..COLL_USER10
<i>Default:</i>	COLL_LOW
<i>S/W Version:</i>	1.0

Description: It represents the sensitivity level (low, medium, high, user-defined) that is used, during the execution of a program, for detecting the collision. A table element of **\$ARM_SENSITIVITY: Arm collision sensitivity** and of **\$COLL_SOFT_PER: Collision compliance percentage** exists for each of these values. A set of predefined constants defines the values this variable can assume:

- COLL_DSBL
- COLL_LOW,
- COLL_MEDIUM,
- COLL_HIGH,
- COLL_MANUAL,
- COLL_USER1 ... COLL_USER10.

In the PROGR state, the COLL_MANUAL sensitivity is always used (no info is prompted to the user), no matter what is the current value of \$COLL_TYPE. If one of the values COLL_USER1 .. COLL_USER10 is assigned to \$COLL_TYPE, it is responsibility of the user to initialise the corresponding table of **\$ARM_SENSITIVITY: Arm collision sensitivity**.

12.71 \$COND_MASK: PATH segment condition mask

Memory category static

Load category: not saved

Data type: integer

Attributes: limited access, field node, WITH MOVE ALONG

Limits none

Default:

S/W Version: 1.0

Description: Each PATH contains a condition handler table (COND_TBL) of 32 INTEGERs. This table is used to specify which condition handlers will be used during the PATH motion. The standard node field \$COND_MASK is a bit oriented INTEGER to determine which of the condition handlers in the COND_TBL should be locally enabled for the segment. For example, if the COND_TBL elements 1, 2, and 3 contain condition handler numbers 10, 20, and 30 respectively, setting \$COND_MASK to 5 for a node will cause condition handlers 10 and 30 to be locally enabled for the segment in a similar way as a MOVE TO ... WITH CONDITION (this is because a value of 5 has bits 1 and 3 set to 1, so COND_TBL[1] and COND_TBL[3] will be enabled.)

```

PROGRAM pth
TYPE nd = NODEDEF
      $MAIN_POS
      $MOVE_TYPE
      $COND_MASK
      i : INTEGER
      b : BOOLEAN
ENDNODEDEF
-- The nodes of this path should either be taught or NODE_APPended
VAR p : PATH OF nd
BEGIN
  CONDITION[10] :
    WHEN TIME 10 AFTER START DO
      .....
  ENDCONDITION

```

```

CONDITION[30] :
    WHEN TIME 20 BEFORE END DO
    .....
    ENDCONDITION
CONDITION[20] :
    WHEN AT START DO
    .....
    ENDCONDITION
.....
    p.COND_TBL[1] := 10 -- Initialization of COND_TBL
    p.COND_TBL[2] := 20
    p.COND_TBL[3] := 30
-- on node 1, condition 10 and 30 will trigger, as 5 is equal
-- to bit 1 and 3, and elements 1 and 3 of COND_TBL
-- contain number 10 and 30.
    p.NODE[1].$COND_MASK := 5
-- on node 4, condition 20 will trigger
    p.NODE[4].$COND_MASK := 2
CYCLE
...
    MOVE ALONG p
...
END p

```

12.72 \$COND_MASK_BACK: PATH segment condition mask in backwards

<i>Memory category</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	Each PATH contains a condition handler table (COND_TBL) of 32 INTEGERS. This table is used for specifying which condition handlers will be used during the PATH motion. The standard node field \$COND_MASK_BACK is a bit oriented INTEGER for determining which of the condition handlers in the COND_TBL should be locally enabled for the segment if the PATH is being interpreted backwards. For example, if the COND_TBL elements 1, 2, and 3 contain the condition handler numbers 10, 20, and 30 respectively, setting \$COND_MASK_BACK to 6 for a node will cause condition handlers 20 and 30 to be locally enabled for the segment (this is because a value of 6 has bits 2 and 3 set to 1, so COND_TBL[2] and COND_TBL[3] will be enabled.)

12.73 \$CONV_ACT: Conveyor Act

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	integer

Attributes: none
Limits 0..10
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_ACT represents the conveyor table which is currently active on the arm. If zero, it indicates there isn't any active tracking on the arm. It can be used to monitor the process.

12.74 \$CONV_BEND_ANGLE: Conveyor Bend Angle

Memory category field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_BEND_ANGLE represents the angle of the conveyor expressed in radians in case of bending conveyor type. It can be used to monitor the process only in tracking mode.

12.75 \$CONV_DIST: Conveyor shift in micron

Memory category field of crnt_data
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_DIST represents the shift of the conveyor truck frame with respect to the zero position (conveyor base frame). It is expressed in micron. As it is an INTEGER variable, \$CRNT_DATA[n].CONV_DIST is mainly useful inside the CONDITIONS.

12.76 \$CONV_ENC: Conveyor Encoder Position

Memory category field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: none

Limits none
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_ENC represents the encoder position of the conveyor expressed in rounds. Can be used to monitor the process.

12.77 \$CONV_SHIFT: Conveyor shift in mm

Memory category field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_SPD represents the velocity of the conveyor expressed in meters per second. Can be used to monitor the process.

12.78 \$CONV_SPD: Conveyor Speed

Memory category field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: WITH MOVE
Limits none
Default:
S/W Version: 1.0
Description: The \$CRNT_DATA[n].CONV_DIST represents the shift of the conveyor truck frame with respect to the zero position (conveyor base frame). It is expressed in micron. As it is an INTEGER variable, \$CRNT_DATA[n].CONV_DIST is mainly useful inside the CONDITIONS.

12.79 \$CRC_IMPORT: Directory for IMPORT

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Directory in which to find files used within IMPORT statement

12.80 \$CRC_RULES: C5G Save & Load rules

Memory category static

Load category: controller. *Minor category* - environment

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is an array of 8 elements with each element representing the definition of a save / load. Each element can be used defining what gets saved/loaded to/from a .C5G file when using Configure Save or Configure Load. The format of the string is important and must adhere to the following rules

Title | <rule>;<rule2>...

Where <rule> contains the following comma separated

- <Parent_type> eg. ARM_DATA
- <Parent_mask> eg. 5 for arm 1 & 3
- <Data_type> eg. INTEGER
- <Category_major> eg. ARM
- <Category_minor> eg. Dyn
- <1D array spec> eg. 7,1
- <2D array spec> eg. 0,0,7,1
- <Wildcard name> eg. CIO*

Example:

```
$CRC_RULES [2] :=  
"TFB|,,,,-1,,,,,,TOOL;,,,,-1,,,,,,UFRAME;,,,,-1,,,,,,BASE"
```

This rule titled 'TFB' has three sub-rules and when used saves/loads the value of \$TOOL, \$BASE and \$UFRAME for all arms.

```
$CRC_RULES [5] := "fred|ARM_DATA,1,AREA,-1,-1,1,1,1,4,1,2,L3D_*
```

This assignment means that a rule called 'fred' is defined including all (-1, -1 mean all major and minor categories) system variables of \$ARM_DATA that are of Array_of_reals datatype, with name L3D_*, array element 1 and array elements [1..4, 1..2].

Such a rule will then include:

```
$ARM_DATA[1].L3D_MASTER[1,1]
$ARM_DATA[1].L3D_MASTER[1,2]
$ARM_DATA[1].L3D_MASTER[2,1]$ARM_DATA[1].L3D_MASTER[2,2]
$ARM_DATA[1].L3D_MASTER[3,1]
$ARM_DATA[1].L3D_MASTER[3,2]
$ARM_DATA[1].L3D_MASTER[4,1]
$ARM_DATA[1].L3D_MASTER[4,2]
$ARM_DATA[1].L3D_XECAM[1]
$ARM_DATA[1].L3D_YECAM[1]
$ARM_DATA[1].L3D_ZECAM[1]
```

12.81 \$CRNT_DATA: Current Arm data

Memory category dynamic
Load category: not saved
Data type: array of crnt_data of one dimension
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: \$CRNT_DATA is an array of predefined records with one element for each arm. The fields of each \$CRNT_DATA element represent the current arm related data

12.82 \$CUSTOM_CNTRL_ID: Identifier for the Controller

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: This string element contains the identification name assigned by the user to the Controller

12.83 \$CUSTOM_ID: Identifier for the arm

Memory category static
Load category: arm *Minor category - configuration*
Data type: array of string[33] of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Each element is a string containing the identification name that the user eventually assigned to an arm.
There is one array element for each arm.

12.84 \$CYCLE: Program cycle count

Memory category program stack

Load category: not saved
Data type: integer
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: It represents a counter for the number of cycles the program has executed. If the CYCLE option is not on the [BEGIN Statement](#), the initial value is 0. Otherwise, it is 1. It is incremented each time a program [END Statement](#) or [EXIT CYCLE Statement](#) is executed. This variable can be useful for initialization.

```

PROGRAM tcycle
ROUTINE do_clean_gun EXPORTED FROM gun
BEGIN
  -- $CYCLE value is 0
  CYCLE
  -- $CYCLE is incremented every CYCLE
  IF $CYCLE MOD 10 = 0 THEN
    WRITE (CYCLE=,$CYCLE,NL)
    do_clean_gun
  ENDIF
END tcycle
  
```

12.85 \$DB_MSG: Message Database Identifier

Memory category static
Load category: controller. *Minor category* - shared
Data type: integer
Attributes: privileged read-write
Limits none
Default: 0x100FE01
S/W Version: 1.0
Description: It is the identifier for the Message Database

12.86 \$DELTA_POS: offsets on final position in the specified (\$DELTA_POS_IN) frame

Memory category field of arm_data
Load category: arm *Minor category* - chain
Data type: position
Attributes: property; WITH MOVE; MOVE ALONG
Limits none
Default: none
S/W Version: 1.15

Description: It can be used for defining the offsets that will influence next motions towards final points of POSITION data type.
\$DELTA_POS_IN: X, Y, Z offsets on final position predefined variable defines the proper frame of reference to be used in such movements.

12.87 \$DELTA_POS_IN: X, Y, Z offsets on final position

Memory category field of arm_data
Load category: arm *Minor category - chain*
Data type: integer
Attributes: property; WITH MOVE; MOVE ALONG
Limits none
Default: none
S/W Version: 1.15
Description: This predefined variable defines the reference frame (NO_DELTA_POS, TOOL, BASE, UFRAME) to be used during next movements affected by **\$DELTA_POS: offsets on final position in the specified (\$DELTA_POS_IN) frame** predefined variable.

12.88 \$DEPEND_DIRS: Dependancy path

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Represents the path of directories to search when using memory load command with dependancies option specified. For example, if \$DEPEND_DIR has the value of "UD:\\user", this directory will be searched upon MEMORY LOAD/DEPEND for searching the dependency files. More directories can be specified and in this case they should be separated by a ". ". The order of insertion of a data determines the priority given to searching phase.

12.89 \$DFT_ARM: Default arm

Memory category static
Load category: controller. *Minor category - environment*
Data type: integer
Attributes: privileged read-write
Limits none
Default:

S/W Version: 1.0
Description: It represents the default arm of the system. It is used in a multi-arm system when the PROG_ARM attribute has not been specified in the PROGRAM header statement. The range of values is from 1 up to 32

12.90 \$DFT_DV: Default devices

Memory category static
Load category: controller. *Minor category* - environment
Data type: array of string of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the various default devices used by the Controller. It is an array of 14 elements having the following meaning:
[1]: Default standard output for PDL2 ('TP:')
[2]: Default system device ('UD:')
[3]: Default backup/restore device ('XD:')
[4]: Default device for non retentive files ('TD:')
[5]: Default communication device ('COM1:')
[6]: Default device for PDL2 ('COMP:')
[7]: Default device for installation (Filer Utility Install) ('COMP:')
[8]: Default device for COMP ('NET0:')
[9]: Default device/directory for configuration ('UD:\sys\cfg')
[10]: Default device for utilities ('UD:\sys\util')
[11]: Default device for user ('UD:\usr')
[12]: Default directory for libraries ('UD:\lib')
[13]: Default temporary directory ('UD:\temp')
[14]: Default device/directory for configuration tools ('UD:\sys\cfg\pdll')

12.91 \$DFT_LUN: Default LUN number

Memory category static
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: LUN_TP
S/W Version: 1.0

Description: It represents the default Logical Unit Number (LUN) to be used for READ and WRITE operations in PDL2 programs. Reasonable values for this variable are identified by LUN_CRT, LUN_TP and LUN_NULL predefined constants. The default value is LUN_TP. When working on the PC video/keyboard, using the WinC5G program, this variable should be set to LUN_CRT for directing by default the serial input/output to the CRT video-keyboard of the PC

12.92 \$DFT_SPD: Default devices speed

Memory category static
Load category: controller. *Minor category* - environment
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: *It contains the default speeds for some controller devices. It is an array of 4 elements having the following meaning:*
[1] reserved
[2] Default speed for file transmission over protocol file transfer by PC (PCFT upon UCMC that mounts WinC5G)
[3..4] reserved

12.93 \$DIN: Digital input

Memory category port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the set of digital input points. The number of available input points depends on the number of I/O boards installed in the system. For further information see also [par. 5.2.1 \\$DIN and \\$DOUT on page 104](#).

12.94 \$DNS_DOMAIN: DNS Domain

Memory category field of board_data
Load category: controller. *Minor category* - shared
Data type: string
Attributes: none
Limits none
Default:

S/W Version: 1.0

Description: This is the name of the DNS Domain from where IP address will be located

12.95 \$DNS_ORDER: DNS Order

Memory category field of board_data

Load category: controller *Minor category - shared*

Data type: integer

Attributes: none

Limits 0..3

Default: 0

S/W Version: 1.0

Description: Order in which addresses are searched:

1: Local host table

2: DNS server first

3: DNS server only

12.96 \$DNS_SERVERS: DNS Servers

Memory category field of board_data

Load category: controller. *Minor category - shared*

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: An array that can contain up to 3 IP addresses for the location of the DNS server

12.97 \$DOUT: Digital output

Memory category port

Load category: not saved

Data type: array of boolean of one dimension

Attributes: pulse usable

Limits none

Default:

S/W Version: 1.0

Description: It represents the set of digital output points. The number of output points available is dependent on the number of I/O boards installed in the system. For further information see also [par. 5.2.1 \\$DIN and \\$DOUT on page 104](#).

12.98 \$DV_STS: the status of DV_CTRL calls

Memory category static
Load category: not saved
Data type: integer
Attributes: none
Limits 0..0
Default: 0
S/W Version: 1.0
Description: This variable contains the status of the last DV_CTRL operation

12.99 \$DV_TOUT: Timeout for asynchronous DV_CTRL calls

Memory category static
Load category: controller. *Minor category* - environment
Data type: integer
Attributes: none
Limits 0..MAXINT
Default: 0
S/W Version: 1.0
Description: This variable contains the timeout value for asynchronous DV_CTRL calls with a value of zero meaning an indefinite wait

12.100 \$DYN_COLL_FILTER: Dynamic Collision Filter

Memory category field of arm_data
Load category: arm. *Minor category* - collision, dynamic
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This array represent the coefficient of the filter that is applied on the residual of current, in the new dynamic model algorithm, in order to detect collision

12.101 \$DYN_DELAY: Dynamic model delay

Memory category field of arm_data

Load category: arm. *Minor category - dynamic*
Data type: integer
Attributes: privileged read-write
Limits 4..100
Default: 4
S/W Version: 1.0
Description: It represents the time in milliseconds that simulates and balances the delays in the computation chain and the propagation of the signal between the trajectory generator and the dynamic model.

12.102 \$DYN_FILTER2: Dynamic Filter for dynamic model

Memory category field of arm_data
Load category: arm
Data type: array of real of two dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It contains three filters for the dynamic model.
One filter (maximum of second order) is used for giving a limit to the band of input signals to the dynamic model block; this filter simulates and balances the phase delay inserted by the microinterpolation serial filters.
The second one (maximum of second order) is used for limiting the action band of the feed forward of the current.
The third one (maximum of second order) is used for synchronizing the feed forward of the current that detects the collisions.

12.103 \$DYN_GAIN: Dynamic gain in inertia and viscous friction

Memory category field of arm_data
Load category: arm. *Minor category - dynamic*
Data type: real
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the gain that allows to tune the contribution in terms of inertia and of axes viscous friction in the current component of the dynamic model.

12.104 \$DYN_MODEL: Dynamic Model

Memory category: field of arm_data
Load category: arm. *Minor category* - dynamic
Data type: array of real of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: This is a set of data for the new dynamic model algorithm that represents the robot identification parameters without payload

12.105 \$DYN_WRIST: Dynamic Wrist

Memory category: field of arm_data
Load category: arm. *Minor category* - dynamic
Data type: array of real of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: This is a set of data for the new dynamic model algorithm that represents the wrist identification parameters without load

12.106 \$DYN_WRISTQS: Dynamic Theta

Memory category: field of arm_data
Load category: arm. *Minor category* - dynamic
Data type: array of real of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: This is a set of data for the new dynamic model algorithm that represents the wrist identification parameters without load calculated during the identification program at the lowest speed

12.107 \$EMAIL_INT: Email integer configuration

Memory category: static
Load category: controller. *Minor category* - unique

Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description:

- [1]: flags
 - 0x01: Feature is enabled
 - 0x04: use APOP authentication
 - 0x10: Use of memory or file system for incoming mail
 - 0x020: Do not remove the directory where attachments are stored upon system startup
- [2]: Maximum size in bytes for incoming messages (if 0, the default is 300K, that is 300 * 1024 bytes)
- [3]: Polling interval for POP3 server (if 0, the default and minimum used value is 60000 ms)
- [4]: Size to reserve to the body of the email (if 0 the default value is 5K, that is 5 * 1024 bytes)
- [5]: timeout (if 0 the default value is 10000 ms)

12.108 \$EMAIL_STR: Email string configuration

Memory category static
Load category: controller. *Minor category* - unique
Data type: array of string of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It is an array of 6 elements having the following meaning:
[1]: POP3 server address or name for the incoming email
[2]: SMTP server address or name for the outgoing email
[3]: sender e-mail address ("From" field of the outgoing e-mail)
[4]: login of the POP3 server
[5]: password of the POP3 server
[6]: directory where attachments are saved. It will be a subdirectory of UD:\SYS\EMAIL

12.109 \$ERROR: Last PDL2 Program Error

Memory category program stack
Load category: not saved
Data type: integer
Attributes: none
Limits 0..0

Default:

S/W Version: 1.0

Description: This variable contains the last error (if any) that occurred in the program.

12.110 \$ERROR_DATA: Last PDL2 Program Error Data

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits 0..0

Default:

S/W Version: 1.0

Description: Additional data associated to the current \$ERROR. This could include the “secondary” error code reported in an alarm.

12.111 \$EXE_HELP: Help on Execute command

Memory category static

Load category: controller. *Minor category - environment*

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: This array can be set by the user to contain those statements that are more frequently needed in the Execute command of the system menu. These statements are displayed in the help window associated to the HELP key pressure of the Execute command together with other statements predefined in the system. This value has only effect if it is saved in the configuration file and the controller is restarted

12.112 \$FDIN: Functional digital input

Memory category port

Load category: not saved

Data type: array of boolean of one dimension

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: It represents the functional digital input points. For further information see also [\\$FDIN](#) and [\\$FDO](#).

12.113 \$FDO: Functional digital output

Memory category port

Load category: not saved

Data type: array of boolean of one dimension

Attributes: pulse usable

Limits: none

Default:

S/W Version: 1.0

Description: It represents the functional digital output points. For further information see also [\\$FDIN](#) and [\\$FDO](#).

12.114 \$FEED_VOLTAGE: Feed Voltage

Memory category static

Load category: not saved

Data type: array of real

Attributes: read-only

Limits: 0..10

Default:

S/W Version: 1.0

Description: This variable contains the values of the current absorptions, in ampere, of the 24 Volt supply outputs.

The meaning of its elements is the following:

- [1] V24 INT supply current (limit 5A) internally used for TP and Options.
- [2] V24 I/O and V24 SAFE outputs common supply current (limit 8A) used by standard and user applications.
- [3] V24 SAFE supply current (in Ampere) used by standard and user applications. The supply output is switched in safety

with the state of the Robot. (DRIVE-ON: power present, DRIVE-OFF: power absent).

12.115 \$FL_ADLM: Array of delimiters

Memory category static

Load category: not saved

Data type: string

Attributes: limited access, WITH OPEN FILE

Limits: none

Default:

S/W Version: 1.0

Description: It represents an array of ASCII delimiting characters for asynchronous READs. This can be changed by setting this variable in the WITH clause of the [OPEN FILE Statement](#). The delimiter character received is placed in the read buffer. A maximum of 16 characters can be specified as delimiters and if \$FL_PASSALL is not TRUE, then the ENTER and down arrow keys are differentiated.

12.116 \$FL_ARM_CNFG: Arm configuration file name

Memory category static
Load category: arm *Minor category - configuration*
Data type: string
Attributes: property
Limits none
Default:
S/W Version: 1.0
Description: It is used to handle the name of the Robot data file, used as information to be displayed and to search for the new file to be used for updating.

12.117 \$FL_AX_CNFG: Axes configuration file name

Memory category static
Load category: arm *Minor category - configuration*
Data type: array of string of two dimensions?
Attributes: privileged read-write, property
Limits none
Default:
S/W Version: 1.0
Description: It is used to handle the name of the Auxiliary Axes data file, used as information to be displayed and to search for the new file to be used for updating.

12.118 \$FL_BINARY: Text or character mode

Memory category static
Load category: not saved
Data type: boolean
Attributes: limited access, WITH OPEN FILE
Limits none
Default:
S/W Version: 1.0
Description: It represents the data format for files. The default data format is text. This can be changed to binary by setting this variable to TRUE in the WITH clause of the

OPEN FILE Statement:

```
OPEN FILE lunid ('FRED.DAT', 'R') WITH $FL_BINARY=TRUE
```

12.119 \$FL_CNG: Configuration file name

Memory category: static
Load category: controller *Minor category* - configuration
Data type: string
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It contains the name of the configuration file.

12.120 \$FL_COMP: Compensation file name

Memory category: field of arm_data
Load category: arm *Minor category* - configuration
Data type: string
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the compensation file name used by the compensation algorithm. There is one of this variable for each arm. The user can assign to this variable the name of the compensation file to be used for that arm. The name should NOT include the file extension and the device specification. If this variable is not initialized or is set to the null string (""), the algorithm looks in UD: for a file with extension .ROB and with the arm number as last character of the file name. Note that, if more than one file are present in UD: with this characteristic, the first one (non necessarily in alphabetical order) is taken. For disabling the algorithm, it is needed to set this variable to the null string and to remove the .ROB file from the UD:. The system returns an error only in case \$FL_COMP is set to a certain value (different from NULL string) and the corresponding file is not present in UD:

12.121 \$FL_DLMT: Delimiter specification

Memory category: static
Load category: not saved
Data type: integer
Attributes: limited access, WITH OPEN FILE
Limits: -1..255
Default:

S/W Version: 1.0
Description: It represents the ASCII delimiting character used when reading data. The default is line feed or carriage return. This can be changed by setting \$FL_DLMT in the WITH clause of the [OPEN FILE Statement](#)

12.122 \$FL_ECHO: Echo characters

Memory category static
Load category: not saved
Data type: boolean
Attributes: limited access, WITH OPEN FILE
Limits none
Default:
S/W Version: 1.0
Description: It represents whether the input characters to a READ are echoed to the corresponding output device or not. For window devices this has a default value of TRUE. This can be changed by setting this variable in the WITH clause of the [OPEN FILE Statement](#)

12.123 \$FL_NUM_CHARS: Number of chars to be read

Memory category static
Load category: not saved
Data type: integer
Attributes: limited access, WITH OPEN FILE
Limits 1..512
Default:
S/W Version: 1.0
Description: It represents the number of characters to be read. This can be changed by setting this variable in the WITH clause of the [OPEN FILE Statement](#)

12.124 \$FL_PASSALL: Pass all characters

Memory category static
Load category: not saved
Data type: boolean
Attributes: limited access, WITH OPEN FILE
Limits none
Default:
S/W Version: 1.0
Description: It represents whether all characters are passed through to the READ. This can be changed by setting this variable in the WITH clause of the [OPEN FILE Statement](#).

This is useful when trying to read all keyboard input.

12.125 \$FL_RANDOM: Random file access

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	limited access, WITH OPEN FILE
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the access method to a file. The normal method is sequential. This can be changed to random access by setting this variable to TRUE in the WITH clause of the OPEN FILE Statement . It is important to set this if FL_GET_POS or FL_SET_POS are to be used

12.126 \$FL_RDFLUSH: Flush on reading

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	limited access, WITH OPEN FILE
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents whether the input buffer is flushed before a READ is issued. This can be changed by setting this variable in the WITH clause of the OPEN FILE Statement . This is useful when reading from a serial device or from a window when there is to be no type-ahead.

12.127 \$FL_STS: Status of last file operation

<i>Memory category:</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	0..0
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the status of the last file operation undertaken by the program.

12.128 \$FL_SWAP: Low or high byte first

Memory category static
Load category: not saved
Data type: boolean
Attributes: limited access, WITH OPEN FILE
Limits none
Default:
S/W Version: 1.0
Description: It represents whether the high byte or the low byte is written to a LUN first. This can be changed by setting this variable in the WITH clause of the [OPEN FILE Statement](#).

12.129 \$FLOW_TBL: Flow modulation algorithm table data

Memory category dynamic
Load category: not saved
Data type: array of \$flow_tbl of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: \$FLOW_TBL is an array of flow modulation schedules with each schedule containing the .FW_xxx fields.

12.130 \$FLY_DBUG: Cartesian Fly Debug

Memory category field of crnt_data
Load category: not saved
Data type: integer
Attributes: none
Limits 0..0
Default: 0
S/W Version: 1.0
Description: It contains a numerical code useful for testing the behavior of the cartesian fly. It is only used when [\\$FLY_TYPE: Type of fly motion](#) is set to FLY_CART. The user can only reset its value to zero. All other possible values are set by the controller.

12.131 \$FLY_DIST: Distance in fly motion

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: property, WITH MOVE; MOVE ALONG
Limits: none
Default: 5
S/W Version: 1.0
Description: It represents the distance, expressed in millimeters, for the trajectory planning in cartesian fly. It is only used when \$FLY_TYPE is set to FLY_CART. This parameter has different meanings depending on the current value of \$FLY_TRAJ. The default value is 5 millimeters.

12.132 \$FLY_PER: Percentage of fly motion

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE; MOVE ALONG
Limits: 1..100
Default: 100
S/W Version: 1.0
Description: It represents the percentage in time of overlapping between the deceleration of a MOVEFLY motion and the acceleration of the next motion. It is used in fly between joint movements, while on cartesian fly it has effects only if \$FLY_TYPE is set to FLY_NORM.

12.133 \$FLY_TRAJ: Type of control on cartesian fly

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE; MOVE ALONG
Limits: FLY_AUTO..FLY_TOL
Default: FLY_TOL
S/W Version: 1.0
Description: It indicates the meaning of [\\$FLY_DIST: Distance in fly motion](#) for the trajectory planning in cartesian fly. It is only used when [\\$FLY_TYPE: Type of fly motion](#) is set to FLY_CART. The following predefined constants can be used to set the value of \$FLY_TRAJ: FLY_AUTO, FLY_TOL, FLY_PASS, FLY_FROM. The default value is FLY_TOL, which forces the distance from the trajectory to the fly point to be less than or equal to the distance set in [\\$FLY_DIST: Distance in fly motion](#).

12.134 \$FLY_TYPE: Type of fly motion

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE; MOVE ALONG
Limits: FLY_NORM..FLY_CART
Default: FLY_NORM
S/W Version: 1.0
Description: It represents the type of Cartesian fly motion. Valid values are represented by the predefined constants FLY_NORM and FLY_CART.

12.135 \$FMI: Flexible Multiple Analog/Digital Inputs

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an analog port array seen as a set of digital ports. It allows the user to group data coming from the Fieldbus. It is useful to group any not aligned information which is to be treated as analog data.
 The information is available at bit level. It is not based upon any other I/O port. The user is allowed to assign it any data of variable length, from 1 to 32 bits.
 For further information see also [\\$FMI](#) and [\\$FMO](#).

12.136 \$FMO: Flexible Multiple Analog/Digital Outputs

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an analog port array seen as a set of digital ports. It allows the user to group data going to the Fieldbus. It is useful to group any not aligned information which is to be treated as analog data.
 The information is available at bit level. It is not based upon any other I/O port. The user is allowed to assign it any data of variable length, from 1 to 32 bits.

For further information see also [par. 5.2.4 \\$FMI and \\$FMO on page 104](#).

12.137 \$FOLL_ERR: Following error

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the current following error between the actual robot position and the desired robot position. The value is expressed in motor turns.

12.138 \$FUI_DIRS: Installation path

<i>Memory category</i>	static
<i>Load category:</i>	controller <i>Minor category</i> - environment
<i>Data type:</i>	string
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	"UD:\inst\"
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the path of directories to search when using the Filer Utility Install command

12.139 \$FW_ARM: Arm under flow modulation algorithm

<i>Memory category</i>	field of \$flow_tbl
<i>Load category:</i>	controller <i>Minor category</i> - flow
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It is the arm which the FLOW MODULATE algorithm should apply to.

12.140 \$FW_AXIS: Axis under flow modulation algorithm

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: integer
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is the axis which the algorithm should apply to. It must be specified only when \$FLOW_TBL[index].FW_VAR is set to 2.

12.141 \$FW_CNVRSN: Conversion factor in case of Flow modulation algorithm

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: real
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Conversion factor. It determines the slope of the flow line. This value can be changed meanwhile the algorithm is working.

12.142 \$FW_ENBL Flow modulation algorithm enabling indicator

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: boolean
Attributes: privileged access
Limits: none
Default:
S/W Version: 1.0
Description: It indicates whether the Flow Modulation algorithm is currently enabled

12.143 \$FW_FLOW_LIM Flow modulation algorithm flow limit

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of two elements indicating the minimum and the maximum values for the flow that define a range out of which the nearest limit is applied.

12.144 \$FW_SPD_LIM Flow modulation algorithm speed limits

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: array of real of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 2 REAL values indicating the minimum and the maximum speed values that define a range out of which the nearest limit is applied.

12.145 \$FW_START Delay in flow modulation algorithm application after start

Memory category: field of \$flow_tbl
Load category: controller *Minor category* - flow
Data type: integer
Attributes: privileged access
Limits: none
Default:
S/W Version: 1.0
Description: It is the time interval (in milliseconds) between the first reading of the speed value (immediately after [FLOW_MOD_ON Built-In Procedure](#)) and the writing of data on the analogue port (specified in [FLOW_MOD_ON Built-In Procedure](#)).

12.146 \$FW_VAR: flag defining the variable to be considered when flow modulate is used

Memory category: field of \$flow_tbl
Load category: controller
Data type: integer
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: integer flag that identifies the variable considered by the flow modulation alghorithm:
1 for \$ARM_VEL (cosmetic sealing),
2 for \$RAD_VEL (glueing).

12.147 \$GEN_OVR: General override

Memory category: static
Load category: not saved
Data type: integer
Attributes: read-only
Limits: 1..100
Default: 100
S/W Version: 1.0
Description: This is an overall percentage value used for controlling the speed, acceleration, and deceleration of each arm in a coordinated manner. This is a system-wide value, applied to all arms.

12.148 \$GI: General System Input

Memory category: port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the General System Input points. For further information see also [par. 5.3.3 \\$GI and \\$GO on page 109](#).

12.149 \$GO: General System Output

Memory category: port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the General System Output points. For further information see also [par. 5.3.3 \\$GI and \\$GO on page 109](#).

12.150 \$GUN: Electrical welding gun

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of string of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 3 strings. Each element contains the name of the electrical welding gun mounted on the correspondent auxiliary axis. The index of the \$GUN element corresponds to the number of the auxiliary axis in the same way as the index of the auxiliary axis in an XTNDPOS variable. The FAMILY tool initializes this variable when the electrical welding gun should be handled.

12.151 \$HAND_TYPE: Type of hand

Memory category: field of arm_data
Load category: input/output. *Minor category* - hand
Data type: array of integer of two dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the type of hand. Each ARM has two hands. Certain aspects such as the type of hand (SINGLE, LINE, PULSE and DUAL), the digital output connections, and the timing values, can be set using this variable.

12.152 \$HDIN: High speed digital input

<i>Memory category</i>	port
<i>Load category:</i>	not saved
<i>Data type:</i>	array of boolean of two dimensions
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents the high speed digital input points: there are two of them for each Arm. This input can be used to quickly read the position of the robot Arm. It can also be used as a general purpose digital input to lock an Arm. Refer to HDIN_READ Built-In Procedure and HDIN_SET Built-In Procedure.</p> <p>Refer also to par. 5.3.5 \$HDIN on page 117 for further information.</p>

12.153 \$HDIN_SUSP: temporarily disables the use of \$HDIN

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	0..1
<i>Default:</i>	0
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>This variable is used for temporarily disabling, when set to 1, what defined for \$HDIN by means of a previous call to the HDIN_SET Built-In Procedure.</p> <p>For re-enabling the \$HDIN monitoring (if it was previously enabled), \$HDIN_SUSP should be set to 0. Note that, when the HDIN_SET Built-In Procedure is called, the \$HDIN_SUSP is automatically set to 0. It is therefore suggested to use the \$HDIN_SUSP only after calling HDIN_SET Built-In Procedure.</p>
<i>Example:</i>	<code>\$CRNT_DATA [1].HDIN_SUSP [2] := 0</code>

12.154 \$HLD_DEC_PER: Hold deceleration percentage

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	100..400
<i>Default:</i>	100

S/W Version: 1.0

Description: It represents the percentage of the deceleration upon HOLD or LOCK for each axis. A value of 100 means the same deceleration profile as that set in \$MTR_DEC_TIME. A value of 200 means half the time is taken for deceleration.

12.155 \$HOME: Arm home position

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: jointpos

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It represents the home position of an arm.

12.156 \$ID_ADRS: Address on the Fieldbus net

Memory category field of io_dev_data

Load category: not saved

Data type: integer

Attributes: none

Limits none

Default: 0

S/W Version: 1.0

Description: Physical address of the Input/Output Device on the Fieldbus net

12.157 \$ID_APPL: Application Code

Memory category field of io_dev_data

Load category: not saved

Data type: integer

Attributes: none

Limits none

Default: 0

S/W Version: 1.0

Description: Application Package related Code for the current Input/Output Device

12.158 \$ID_DATA: Fieldbus specific data

Memory category: field of io_dev_data
Load category: not saved
Data type: array [10] of integer
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: General data of the Fieldbus net related to the current Input/Output Device

12.159 \$ID_ERR: error Code

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Last error detected on the current Input/Output device

12.160 \$ID_IDX: Device index

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Index of the current Device, in the array

12.161 \$ID_ISIZE: Input bytes quantity

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none

Default: 0
S/W Version: 1.0
Description: Total amount of input bytes of the current Input/Output Device

12.162 \$ID_ISMST: this Device is a Master (or Controller)

Memory category field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Possible values are:
 – 1: device is an i/o device on a fieldbus network
 – 2: device is robot as seen by a PLC

12.163 \$ID_MASK: miscellaneous mask of bits

Memory category field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: It is a bitmask

12.164 \$ID_NAME: Device name

Memory category field of io_dev_data
Load category: not saved
Data type: string [64]
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Input/Output Device unique name

12.165 \$ID_NET: Fieldbus type

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Network type for the current Input/Output Device:
1 - Virtual
2 - Profibus Interface 1
3 - Profibus Interface 2
4 - DeviceNet Interface 1
5 - DeviceNet Interface 2
6 - CanOpen Interface 1
7 - CanOpen Interface 2
8 - Powerlink
9 - Profinet I/O

12.166 \$ID_NOT_ACTIVE: not active Device at net boot time

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: It indicates whether the device is not active on the fieldbus at boot time:
0 - active; 1 - not active

12.167 \$ID_OSIZE: output bytes quantity

Memory category: field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Total amount of output bytes of the current Input/Output Device

12.168 \$ID_STS: Device status

Memory category field of io_dev_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Status of the current Device

12.169 \$ID_TYPE: configuration status

Memory category field of io_dev data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: It indicates the Device configuration status:
 0 - physical
 1 - virtual

12.170 \$IN: IN digital

Memory category port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the set of digital input points reserved for PDL2 applications.
 PDL2 programs written by the end-user should not refer this port in order to avoid conflicts
 in I/O mapping definitions.
 For further information see also [par. 5.2.2 \\$IN and \\$OUT on page 104](#).

12.171 \$IO_DEV: Input/Output Device Table

Memory category port

<i>Load category:</i>	not saved
<i>Data type:</i>	array of record of two dimensions
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>This predefined variable has a dynamic number of elements: it changes every I/O compiling time and can be obtained by means of the \$NUM_IO_DEV: Number of Input/Output Devices predefined variable.</p> <p>Each Input/Output Device has its own element in this table. \$IO_DEV is an array of predefined records with one element for each Input/Output Device.</p> <p>The fields of each record are as follows:</p> <ul style="list-style-type: none"> \$ID_STS: Device status \$ID_IDX: Device index \$ID_TYPE: configuration status \$ID_NET: Fieldbus type \$ID_ADRS: Address on the Fieldbus net \$ID_ISIZE: Input bytes quantity \$ID_ISMST: this Device is a Master (or Controller) \$ID_OSIZE: output bytes quantity \$ID_APPL: Application Code \$ID_NOT_ACTIVE: not active Device at net boot time \$ID_ERR: error Code \$ID_NAME: Device name \$ID_MASK: miscellaneous mask of bits \$ID_DATA: Fieldbus specific data <p>See also DV_CNTRL Built-In Procedure, bit 15, to get the index to access a specific Device, passing the Device name.</p>

12.172 \$IO_STS: Input/Output point StatusTable

<i>Memory category</i>	port
<i>Load category:</i>	not saved
<i>Data type:</i>	array of record of two dimensions
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>This predefined variable has a dynamic number of elements: it changes every I/O compiling time and can be obtained by means of the \$NUM_IO_STS: Number of Input/Output points predefined variable.</p> <p>Each Input/Output Point has its own element in this table. \$IO_STS is an array of predefined records with one element for each I/O Point.</p> <p>The fields of each record are as follows:</p> <ul style="list-style-type: none"> \$IS_STS: Port status

\$IS_DEV: associated Device
 \$IS_PORT_CODE: Port type
 \$IS_PORT_INDEX: Port type index
 \$IS_SIZE: Port dimension
 \$IS_RTN: retentive output flag
 \$IS_PRIV: privileged Port flag
 \$IS_NAME: I/O point name
 \$IS_HELP_STR: User Help string

See also [DV_CNTRL Built-In Procedure](#), bit 14, to get the index to access the status of a specific I/O Point, passing the I/O Point Name.

12.173 \$IPERIOD: Interpolator period

Memory category: static
Load category: controller. *Minor category* - environment
Data type: integer
Attributes: property
Limits: 1..8
Default: 2
S/W Version: 1.0
Description: It represents the time period for the position generation.

12.174 \$IREG: Integer register - saved

Memory category: static
Load category: controller. *Minor category* - vars
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Integer registers that can be used by users instead of having to define variables. Also for Real, String, Boolean, Jointpos, Position and Xtdpos datatypes there are saved and non-saved registers.

12.175 \$IREG_NS: Integer registers - not saved

Memory category: static
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none

Default:

S/W Version: 1.0

Description: Non-saved Integer registers that can be used by users instead of having to define variables. Also for Real, String, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.176 \$IR_TBL: Interference Region table data

Memory category dynamic

Load category: not saved

Data type: array[32] of records of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: \$IR_TBL is an array of 32 elements, with each element containing the \$IR_xxx fields. The fields of each record are as follows:

- \$IR_ARM: Interference Region Arm
- \$IR_CNFG: Interference Region Configuration
- \$IR_ENBL: Interference Region enabled flag
- \$IR_IN_OUT: Interference Region location flag
- \$IR_JNT_N: Interference Region negative joint
- \$IR_JNT_P: Interference Region positive joint
- \$IR_PBIT: Interference Region port bit
- \$IR_PERMANENT: Interference Region permanent
- \$IR_PIDX: Interference Region port index
- \$IR_POS1: Interference Region position
- \$IR_POS2: Interference Region position
- \$IR_POS3: Interference Region position
- \$IR_POS4: Interference Region position
- \$IR_PTYPE: Interference Region port type
- \$IR_REACHED: Interference Region position reached flag
- \$IR_SHAPE: Interference Region Shape
- \$IR_TYPE: Interference Region type
- \$IR_UFRAME: Interference Region Uframe

See also: IR_SET Built-In Procedure, IR_SET_DIG Built-In Procedure, IR_SWITCH Built-In Procedure

12.177 \$IR_ARM: Interference Region Arm

Memory category field of ir_tbl

Load category: controller *Minor category* - Interference Region

Data type: integer

Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: It indicates the arm referred by the \$IR_TBL[ir_index] Interference Region element.

12.178 \$IR_CNFG: Interference Region Configuration

Memory category field of ir_tbl
Load category: not saved
Data type: integer
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: It indicates the internal configuration referred by the \$IR_TBL[ir_index] Interference Region element. Reserved.

12.179 \$IR_CON_LOGIC: Interference Region Consent Logic

Memory category field of ir_tbl
Load category: controller *Minor category* - Interference Region
Data type: boolean
Attributes: none
Limits none
Default: TRUE
S/W Version: 1.0
Description: This flag indicates the logic for the consent to enter Interference region defined by the [IR_SWITCH Built-In Procedure](#).

12.180 \$IR_ENBL: Interference Region enabled flag

Memory category field of ir_tbl
Load category: not saved
Data type: boolean
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0

Description: If \$IR_TBL[ir_index].IR_ENBL is TRUE, it indicates that the \$IR_TBL[ir_index] Interference Region is enabled.

12.181 \$IR_IN_OUT: Interference Region location flag

Memory category field of ir_tbl

Load category: controller *Minor category - Interference Region*

Data type: boolean

Attributes: privileged read-write

Limits none

Default: FALSE

S/W Version: 1.0

Description: This flag indicates the boolean value (ON/OFF) the digital output port must be set to, as soon as the arm enters the \$IR_TBL[ir_index] Interference Region.

12.182 \$IR_JNT_N: Interference Region negative joint

Memory category field of ir_tbl

Load category: controller *Minor category - Interference Region*

Data type: array of real of one dimension

Attributes: none

Limits none

Default: none

S/W Version: 1.0

Description: This is the negative joint defined for the \$IR_TBL[ir_index] element.

12.183 \$IR_JNT_P: Interference Region positive joint

Memory category field of ir_tbl

Load category: controller *Minor category - Interference Region*

Data type: array of real of one dimension

Attributes: none

Limits none

Default: none

S/W Version: 1.0

Description: This is the positive joint defined for the \$IR_TBL[ir_index] element.

12.184 \$IR_PBIT: Interference Region port bit

Memory category field of ir_tbl
Load category: controller *Minor category* - Interference Region
Data type: integer
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: This variable contains the port bit defined for a \$IR_TBL element.

12.185 \$IR_PBIT_RC: Interference Region port bit for reservation and consent

Memory category field of ir_tbl
Load category: controller *Minor category* - Interference Region
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: This array contains the port bit defined for an \$IR_TBL element. Element 1 for reservation, element 2 for consent.

12.186 \$IR_PERMANENT: Interference Region permanent

Memory category field of ir_tbl
Load category: controller *Minor category* - Interference Region
Data type: integer
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: A value of 1 indicates that the \$IR_TBL[ir_index] Interference Region element is defined as permanent.

12.187 \$IR_PIDX: Interference Region port index

Memory category field of ir_tbl

Load category: controller *Minor category - Interference Region*
Data type: integer
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: This variable contains the presence port index defined for a \$IR_TBL element

12.188 \$IR_PIDX_RC: Interference Region port index for reservation and consent

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: This array contains the port index defined for an \$IR_TBL element. Element 1 for reservation, element 2 for consent.

12.189 \$IR_POS1: Interference Region position

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: position
Attributes: none
Limits none
Default: none
S/W Version: 1.0
Description: This is one of the four positions associated to the \$IR_TBL[ir_index] element.

12.190 \$IR_POS2: Interference Region position

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: position
Attributes: none
Limits none
Default: none
S/W Version: 1.0

Description: This is one of the four positions associated to the \$IR_TBL[ir_index] element.

12.191 \$IR_POS3: Interference Region position

Memory category field of ir_tbl

Load category: controller *Minor category* - Interference Region

Data type: position

Attributes: none

Limits none

Default: none

S/W Version: 1.0

Description: This is one of the four positions associated to the \$IR_TBL[ir_index] element.

12.192 \$IR_POS4: Interference Region position

Memory category field of ir_tbl

Load category: controller *Minor category* - Interference Region

Data type: position

Attributes: none

Limits none

Default: none

S/W Version: 1.0

Description: This is one of the four positions associated to the \$IR_TBL[ir_index] element.

12.193 \$IR_PTYPE: Interference Region port type

Memory category field of ir_tbl

Load category: controller *Minor category* - Interference Region

Data type: integer

Attributes: privileged read-write

Limits none

Default: none

S/W Version: 1.0

Description: This variable contains the presence port type defined for a \$IR_TBL element. Reserved.

12.194 \$IR_PTYPE_RC: Interference Region port type for reservation and consent

Memory category field of ir_tbl

Load category: controller *Minor category - Interference Region*
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default: none
S/W Version: 1.0
Description: This array contains the port types defined for a \$IR_TBL element. Element 1 for reservation, element 2 for consent.

12.195 \$IR_REACHED: Interference Region position reached flag

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: boolean
Attributes: Read-only
Limits none
Default: none
S/W Version: 1.0
Description: If \$IR_TBL[ir_index].IR_REACHED is TRUE, it indicates that the arm is inside the Interference Region.

12.196 \$IR_RES_LOGIC: Interference Region reservation logic

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: boolean
Attributes: none
Limits none
Default: TRUE
S/W Version: 1.0
Description: This flag indicates the logic for the reservation of Interference Region defined by the IR_SWITCH Built-In Procedure.

12.197 \$IR_SHAPE: Interference Region Shape

Memory category field of ir_tbl
Load category: controller *Minor category - Interference Region*
Data type: integer
Attributes: privileged read-write

<i>Limits</i>	none
<i>Default:</i>	none
<i>S/W Version:</i>	1.0
<i>Description:</i>	It indicates the geometric shape referred by the \$IR_TBL[ir_index] Interference Region element. The allowed values are: <ul style="list-style-type: none">- IR_NOT_PRESENT (0) = it indicates the Region is not present- IR_PARALLELEPIPED(1) = it indicates the Region is a parallelepiped- IR_SPHERE (2) = it indicates the Region is a sphere- IR_CYLINDER (3) = it indicates the Region is a cylinder- IR_PLANE (4) = it indicates the Region is a plane- IR_JOINT (10) = it indicates the Region is of joint type

12.198 \$IR_TYPE: Interference Region type

<i>Memory category</i>	field of ir_tbl
<i>Load category:</i>	controller <i>Minor category</i> - Interference Region
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	none
<i>S/W Version:</i>	1.0
<i>Description:</i>	It indicates the type of the \$IR_TBL[ir_index] Interference Region element. Allowed values are the following predefined constants: <ul style="list-style-type: none">- IR_FORBIDDEN (0) = it defines the Region type is “forbidden”- IR_MONITORED (1) = it defines the Region type is “monitored”- IR_HYBRID (2) = it defines the Region type is “hybrid”.

12.199 \$IR_UFRAME: Interference Region Uframe

<i>Memory category</i>	field of ir_tbl
<i>Load category:</i>	controller <i>Minor category</i> - Interference Region
<i>Data type:</i>	position
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	none
<i>S/W Version:</i>	1.0
<i>Description:</i>	This is the User frame associated to the \$IR_TBL[ir_index] element.

12.200 \$IS_DEV: associated Device

<i>Memory category</i>	field of io_sts_data
<i>Load category:</i>	not saved

Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Index of the owning Device in the [\\$IO_DEV: Input/Output Device Table](#).

12.201 \$IS_HELP_INT: User Help code

Memory category field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Code of the User Help which describes the associated I/O Point.

12.202 \$IS_HELP_STR: User Help string

Memory category field of io_sts_data
Load category: not saved
Data type: string [64]
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: User Help which describes the associated I/O Point

12.203 \$IS_NAME: I/O point name

Memory category field of io_sts_data
Load category: not saved
Data type: string [64]
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Unique Name of the associated I/O point

12.204 \$IS_PORT_CODE: Port type

Memory category: field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Input/Output point Comau's code:
 1 - DIN
 2 - DOUT
 3 - GI
 4 - GO
 5 - AIN
 6 - AOUT
 9 - SDI
 10 - SDO
 20 - IN
 21 - OUT
 27 - FMI
 28 - FMO

12.205 \$IS_PORT_INDEX: Port type index

Memory category: field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: Port type Index of the associated Input/Output point

12.206 \$IS_PRIV: privileged Port flag

Memory category: field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0

Description: It indicates whether the Input/Output point can be referenced only by a privileged PDL2 program:
– 0 - not privileged - i/o point access is always available;
– 1 - i/o point access requires privilege

12.207 \$IS_RTN: retentive output flag

Memory category field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: It indicates that the Input/Output point is retentive against the power failure recovery

12.208 \$IS_SIZE: Port dimension

Memory category field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Size (in bit) of the Input/Output point

12.209 \$IS_STS: Port status

Memory category field of io_sts_data
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: 0
S/W Version: 1.0
Description: Port status:
1 - physical
2 - virtual
8 - forced.

12.210 \$JERK: Jerk control values

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits: 0..100
Default: 0
S/W Version: 1.0
Description: It represents the percentage of acceleration time and deceleration time used in the constant jerk phase. This is a variation of acceleration
 \$JERK[1]: Determines percentage of acceleration time in the jerk phase
 \$JERK[2]: reserved
 \$JERK[3]: reserved
 \$JERK[4]: Determines percentage of deceleration time in the jerk phase

12.211 \$JL_TABLE: Internal Arm data

Memory category: field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of real of two dimensions
Attributes: privileged read-write
Limits: none
Default: none
S/W Version: 1.0
Description: There are certain fields of \$ARM_DATA which do not have corresponding predefined variables, but instead can be referenced using these general arrays. The format and meaning of the data within these fields is reserved.

12.212 \$JNT_LIMIT_AREA: Joint limits of the work area

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: privileged read-write
Limits: none
Default: none
S/W Version: 1.0
Description: This array represents the joint limits of the work area

12.213 \$JNT_MASK: Joint arm mask

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: none
Default: 0
S/W Version: 1.0
Description: Each bit in the INTEGER represents whether the corresponding axis is present for the arm.

12.214 \$JNT_MTURN: Check joint Multi-turn



NOTE THAT in the following situations

- Spot Welding application with ServoGun, without equalizing functionality,
- any sensor/mechanism performing real-time modification of POSITIONS (e.g. vision systems for robot guiding, SBCU type sensors for TCP modification, etc.),

it is STRONGLY RECOMMENDED to insert the following statement at the beginning of the User PDL2 Program:

```
$jnt_mturn := FALSE
```

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: boolean
Attributes: field node, WITH MOVE; MOVE ALONG
Limits: ON/OFF
Default: ON
S/W Version: 1.0
Description: This variable has effect only on the axes with a range greater than 360 degrees (\$STRK-END_P[axis] - \$STRK-END_N[axis] > 360). If set to FALSE, these axes will be forced to move less than 180 degrees. This variable has effect during joint moves to POSITION and Cartesian moves with \$ORNT_TYPE=WRIST_JNT. The default value is TRUE meaning that the multiturn is allowed

12.215 \$JNT_OVR: joint override

Memory category: field of arm_data
Load category: arm. *Minor category* - overrides
Data type: array of integer of one dimension
Attributes: WITH MOVE; MOVE ALONG

<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the override value for each joint of a robot arm. This variable scales speed, acceleration, and deceleration, so that the trajectory remains constant with changes in the override value. This variable is considered during the planning phase of a motion, therefore it will not affect the current active motion but the next one. The default value is 100%

12.216 \$JNT_SM: joint trajectory planning indicator

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - configuration</i>
<i>Data type:</i>	boolean
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	OFF
<i>S/W Version:</i>	1.0
<i>Description:</i>	It determines which joint trajectory planning method is used. If FALSE, standard method; if TRUE, SmartMove (if the related software option is enabled).

12.217 \$JOG_INCR_DIST: Increment Jog distance

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - configuration</i>
<i>Data type:</i>	real
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	0
<i>S/W Version:</i>	1.0
<i>Description:</i>	It is the distance in millimeters that is undertaken by cartesian moves or by translational axes in joint moves. The default value is 1 millimeter.

12.218 \$JOG_INCR_ENBL: Jog incremental motion

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0

Description: When the value is TRUE, the jog motion starts to be executed in an incremental way. This means that, upon each pressure of one of the jog keys, a motion similar to what specified in \$JOG_INCR_ROT (in case of rotational movements) and \$JOG_INCR_DIST (in case of translational movements) is executed.

12.219 \$JOG_INCR_ROT: Rotational jog increment

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents the evolution in orientation during cartesian moves or the rotations of axes during joint moves. The default value is 0.1 degrees

12.220 \$JOG_SPD_OVR: Jog speed override

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: none
Limits: 1..100
Default: 100
S/W Version: 1.0
Description: It represents the speed override percentage for jogging in cartesian coordinates on a particular arm. There is one value per arm. Acceleration and deceleration are not affected by changes in its value. Changes in \$JOG_SPD_OVR take affect on the next jogging session.

12.221 \$JPAD_DIST: Distance between user and Jpad

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: property
Limits: 0.0.. 25000.0
Default: 0.0
S/W Version: 1.0

Description: It is the distance between the user and the robot base. It is used for defining the Jog Pad system of reference.

12.222 \$JPAD_ORNT:Teach Pendant Angle setting

Memory category field of arm_data.

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: property

Limits -180.0 .. 180.0

Default: 0.0

S/W Version: 1.0

Description: It is the angle in respect with the current system of reference for the Jog Pad

12.223 \$JPAD_TYPE: TP Jpad modality rotational or translational

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: integer

Attributes: property

Limits JPAD_LIN..JPAD_ROT

Default: JPAD_LIN

S/W Version: 1.0

Description: Rotational or translational modality for the usage of the Jog Pad device. Two predefined constants are used for the value: JPAD_LIN (for translation) and JPAD_ROT (for rotational).

12.224 \$JREG: Jointpos registers - saved

Memory category static

Load category: controller. *Minor category* - vars

Data type: array of jointpos of two dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Jointpos registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Position and Xtndpos datatypes there are saved and non-saved registers.

12.225 \$JREG_NS: Jointpos register - not saved

Memory category static
Load category: not saved
Data type: array of jointpos of two dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Non-saved Jointpos registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Position and Xtndpos datatypes there are saved and non-saved registers.

12.226 \$L_ALONG_1D: Internal Loop data

Memory category field of loop_data
Load category: not saved
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
S/W Version: 1.0
Description: There are certain fields of \$LOOP_DATA which format and meaning of the data is reserved.

12.227 \$L_ALONG_2D: Internal Loop data

Memory category field of loop_data
Load category: not saved
Data type: array of integer of two dimension
Attributes: privileged read-write
Limits none
S/W Version: 1.0
Description: There are certain fields of \$LOOP_DATA which format and meaning of the data is reserved.

12.228 \$L_AREAL_1D: Internal Loop data

Memory category field of loop_data

Load category: not saved
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
S/W Version: 1.0
Description: There are certain fields of \$LOOP_DATA which format and meaning of the data is reserved.

12.229 \$L_AREAL_2D: Internal Loop data

Memory category field of loop_data
Load category: not saved
Data type: array of real of two dimension
Attributes: privileged read-write
Limits none
S/W Version: 1.0
Description: There are certain fields of \$LOOP_DATA which format and meaning of the data is reserved.

12.230 \$LAD_OVR: TP Linear Acc/Dec Override for FLY CART

Memory category field of crnt_data
Load category: not saved
Data type: integer
Attributes: none
Limits 1..100
Default: 100
S/W Version: 1.0
Description: It is possible to dynamically decrease the limit of the linear acceleration deceleration parameters.
 This override mainly modulates the linear acceleration and deceleration. It changes [\\$LIN_ACC_LIM: Linear acceleration limit](#) and [\\$LIN_DEC_LIM: Linear deceleration limit](#) with the value of its percentage. It works only for the FLY CART Algorithm.

12.231 \$LAST_JNT_AXIS: Number of the last joint of the arm

Memory category field of arm_data
Load category: not saved
Data type: integer

Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: It represents the number of the last joint of the Arm.

12.232 \$LATCH_CNFG: Latched alarm configuration setting

Memory category static
Load category: controller. *Minor category - environment*
Data type: integer
Attributes: none
Limits none
Default: reserved
S/W Version: 1.0
Description: Different bits are associated to the alarms which are latched and therefore which require user acknowledgement

- Bit 1: a new software has been loaded
- Bit 2: an I/O is forced and simulated
- Bit 3: collision detected
- Bit 4: T2 entry
- Bit 5: AUTO/T2/REMOTE entered with MGD.

 In order to make the error unlatchable, the corresponding bit must be set to 0.

12.233 \$LIN_ACC_LIM: Linear acceleration limit

Memory category field of arm_data
Load category: arm. *Minor category - configuration*
Data type: real
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the maximum linear acceleration expressed in meters per second squared.

12.234 \$LIN_DEC_LIM: Linear deceleration limit

Memory category field of arm_data
Load category: arm. *Minor category - configuration*
Data type: real
Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It represents the maximum linear deceleration expressed in meters per second squared.

12.235 \$LIN_SPD: Linear speed

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: property, WITH MOVE; MOVE ALONG

Limits none

Default:

S/W Version: 1.0

Description: It represents the linear velocity in meters per second for a Cartesian motion. The value is used during the planning phase of a motion when \$SPD_OPT is set to SPD_LIN. In this case the velocity of the TCP corresponds to SPD_LIN less a general override percentage, as long as the speed is within the value of \$LIN_SPD_LIM

12.236 \$LIN_SPD_LIM: Linear speed limit

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It represents the maximum linear speed expressed in meters per second. The value is used by the trajectory planner when [\\$SPD_OPT: Type of speed control](#) is set to SPD_CONST.

12.237 \$LIN_SPD_RT: Run-time Linear speed

Memory category field of arm_data

Load category: arm *Minor category* - overrides

Data type: real

Attributes: property

Limits

Default: 0.0

S/W Version: 1.0

Description: This predefined variable Represents the linear velocity in millimeters per millisecond for a Cartesian motion. The value is used during the planning phase of a motion when [\\$SPD_OPT: Type of speed control](#) is set to SPD_LIN. Its value could be changed run-time, during the motion execution. In this case the velocity of the TCP corresponds to LIN_SPD_RT and will change according its new value less a general override percentage, as long as the speed is within the value of [\\$LIN_SPD_LIM: Linear speed limit](#). Setting this variable to 0 disables this performance and the motion will be executed according to the value of [\\$LIN_SPD: Linear speed](#), setted at the begin of the movement, and the value of [\\$LIN_SPD_RT_OVR: Run-time Linear speed override](#).

12.238 \$LIN_SPD_RT_OVR: Run-time Linear speed override

Memory category field of arm_data
Load category: arm *Minor category - overrides*
Data type: integer
Attributes: property
Limits the maximum value is (\$LIN_SPD_LIM / \$LIN_SPD) * 100
Default: 100
S/W Version: 1.0
Description: This predefined variable is used for changing the linear speed override during the execution of linear and circular motions. It takes effect immediately, also on the running motion.
 For further information about run-time changing the Linear Speed Override, refer to the [Motion Programming](#) manual - par. **Run-Time Speed Override**.

12.239 \$LOG_TO_CHANNEL: Logical to channel relationship

Memory category static
Load category: arm *Minor category - configuration*
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: Represents the mapping between the arm axes and the physical channels.

12.240 \$LOG_TO_DRV: Logical to physical drives relationship

Memory category static

Load category: arm *Minor category* - configuration
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: Represents the mapping between the arm axes and the physical channels to the drives.

12.241 \$LOOP_DATA: Loop data

Memory category static
Load category: not saveds
Data type: array of loop_data of one dimension
Attributes: none
Limits
Default:
S/W Version: 1.0
Description: This predefined variable is an array of predefined records with one element for each Arm. The fields of each \$LOOP_DATA element represent the data related to the control loops for that Arm. Each element is an electro-mechanical peculiarity of the machine and the servo-control parameters.

12.242 \$MAIN_JNTP: PATH node main jointpos destination

Memory category static
Load category: not saved
Data type: jointpos
Attributes: limited access, field node
Limits none
Default:
S/W Version: 1.0
Description: See the description of [\\$MAIN_POS: PATH node main position destination](#)

12.243 \$MAIN_POS: PATH node main position destination

Memory category static
Load category: not saved
Data type: position
Attributes: limited access, field node

Limits none

Default:

S/W Version: 1.0

Description: The standard node fields \$MAIN_POS, \$MAIN_JNTP, and \$MAIN_XTND can be used for defining the type of positional data for the destination of each node of a PATH. Only one such field can be specified in the node definition (NODEDEF).

12.244 \$MAIN_XTND: PATH node main xtndpos destination

Memory category static

Load category: not saved

Data type: xtndpos

Attributes: limited access, field node

Limits none

Default:

S/W Version: 1.0

Description: See the description of [\\$MAIN_POS: PATH node main position destination](#)

12.245 \$MAN_SCALE: Manual scale factor

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: privileged read-write, property

Limits none

Default:

S/W Version: 1.0

Description: It represents the ratio between manual and automatic speed.

12.246 \$MDM_INT: Modem Configuration

Memory category static

Load category: controller. *Minor category* - environment

Data type: array of integer of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is an array having the following meaning:

[1]: Number of rings before modem pickup

[2]: Modem command timeout
[3]: Modem handshake (negotiation) timeout

12.247 \$MDM_STR: Modem Configuration

Memory category static
Load category: controller. *Minor category* - environment
Data type: array of string of one dimension
Attributes: none
Limits none
Default: reserved
S/W Version: 1.0
Description: It is an array having the following meaning:
[1]:modem initialization string

12.248 \$MOD_ACC_DEC: Modulation of acceleration and deceleration

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is a set of data for the algorithm which controls variable acceleration and deceleration based upon the arm position.

12.249 \$MOD_MASK: Joint mod mask

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: Each bit in the INTEGER represents whether the corresponding axis has the modulation algorithm active

12.250 \$MOVE_STATE: Move state

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents the current state of the motion environment. Possible values are:</p> <ul style="list-style-type: none"> - 0 : no move has been issued - 1 : there are pending moves waiting to be executed - 2 : there are moves that have been interrupted and that can be resumed - 8 : there are moves that have been cancelled - 16 : there are active moves - 32 : there is a recovery movement in progress - 64 : there are active moves issued in Jog mode <p>The value of this system variable is dynamically updated. Its value must be read as a bit mask because it is possible that more than one of the listed above situations is active at the same time</p>

12.251 \$MOVE_TYPE: Type of motion

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	WITH MOVE; MOVE ALONG
<i>Limits</i>	JOINT..SEG_VIA
<i>Default:</i>	JOINT
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents the type of interpolation to be used for motion trajectory. Valid values are represented by the predefined constants JOINT, LINEAR and CIRCULAR. In circular moves in paths, \$MOVE_TYPE should be set to SEG_VIA for defining the VIA node.</p>

12.252 \$MTR_ACC_TIME: Motor acceleration time

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0

Description: It represents the minimum motor acceleration time expressed in milliseconds

12.253 \$MTR_CURR: Motor current

Memory category field of crnt_data

Load category: not saved

Data type: array of real of one dimension

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: It represents the actual motor current expressed in amperes

12.254 \$MTR_DEC_TIME: Motor deceleration time

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: array of integer of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It represents the minimum motor deceleration time expressed in milliseconds.

12.255 \$MTR_SPD_LIM: Motor speed limit

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: array of integer of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It represents the maximum motor speed expressed in rotations per minute.

12.256 \$NET_B: Ethernet Boot Setup

Memory category static

Load category: controller. *Minor category* - environment

Data type: array of string of one dimension

Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It is an array of 4 elements having the following meaning:
[1]: IP address of server containing the boot file (e.g. 129.144.32.100)
[2]: User login for boot file (e.g. C5G_USER)
[3]: Password (e.g. C5G_PASS)
[4]: Configuration file (e.g. C5G_CNFG)

12.257 \$NET_B_DIR: Ethernet Boot Setup Directory

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It is the name of directory in which to find the boot file (e.g. /export/home/C5G)

12.258 \$NET_C_CNFG: Ethernet Client Setting Modes

Memory category static
Load category: controller. *Minor category - environment*
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It is an array of 9 elements: the first one is mapped on NET1:, the second one on NET2:, ..., the last one is mapped on NET9: . The bits of each element have the following meaning:
– Bit 1: if set, the transfer is done in Binary mode otherwise in ASCII mode
– Bit 2: if set, some additional information are sent to the user (Verbose)
– Bit 3: if set, the overwrite modality is activated
– Bit 4-8: reserved.

12.259 \$NET_C_DIR: Ethernet Client Setup Default Directory

Memory category: static
Load category: controller. *Minor category - environment*
Data type: array of string of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 9 elements: the first one is mapped on NET1:, the second one on NET2:, ..., the last one is mapped on NET9:. Each element contains the name of the default directory (e.g. /home/fred). If both elements are defined and used, then the directory path must be defined as an absolute path (relative to root or home) and not as a relative path to the last accessed directory. For issuing a FilerView command to the contents of subdir1 that is a subdirectory of dir1 on the remote host of NET1:, should set the variable as \$NET_C_DIR[1] := '\dir1\subdir1'

12.260 \$NET_C_HOST: Ethernet Client Setup Remote Host

Memory category: static
Load category: controller. *Minor category - environment*
Data type: array of string of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 9 elements: the first one is mapped on NET1:, the second one on NET2:, ..., the last one is mapped on NET9:. Each element contains the name of the remote host (e.g. ibm_server).

12.261 \$NET_C_PASS: Ethernet Client Setup Password

Memory category: static
Load category: controller. *Minor category - environment*
Data type: array of string of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0

Description: It is an array of 9 elements: the first one is mapped on NET1:, the second one on NET2:, ..., the last one is mapped on NET9: . Each element contains the name of the password (e.g. Smidth).

12.262 \$NET_C_USER: Ethernet Client Setup Login Name

Memory category static

Load category: controller. *Minor category - environment*

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is an array of 9 elements: the first one is mapped on NET1:, the second one on NET2:, ..., the last one is mapped on NET9: . Each element contains the name of the login (e.g. Pippo)

12.263 \$NET_HOSTNAME: Ethernet network hostnames

Memory category static

Load category: retentive. *Minor category - unique*

Data type: array of string of two dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: \$NET_HOSTNAME

It contains information about hostnames. It is an array of 8 x 2 elements having the following meaning:

[1]: Name of the host eg. sun01

[2]: IP address of this board (e.g. 129.144.32.100)

12.264 \$NET_I_INT: Ethernet Network Information (integers)

Memory category static

Load category: not saved

Data type: array of integer of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It is an array of 2 elements with the following meaning:

[1]: Number of active servers

[2]: reserved

12.265 \$NET_I_STR: Ethernet Network Information (strings)

Memory category static

Load category: not saved

Data type: array of string of two dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It is an array of 2 elements having the following meaning:

[1]: List of hosts connected

[2]: List of user connected

12.266 \$NET_L: Ethernet Local Setup

Memory category static

Load category: retentive. *Minor category* - unique

Data type: array of string of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: This predefined variable contains information about the network settings.

It is an array of 6 elements having the following meaning:

[1]: IP address of this board (e.g. 129.144.32.100)

[2]: Host identifier, that is the name of this board (e.g. robot1).

This element should not contain strings longer than 8 characters for guaranteeing a good functioning of the FTP protocol.

[3]: Subnet mask (e.g. 255.0.0)

[4]: IP address on backplane

[5]: Subnet mask on backplane

[6]: Gateway inet

12.267 \$NET_MOUNT: Ethernet network mount

Memory category static

Load category: retentive. *Minor category* - unique

Data type: array of string of two dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: \$NET_MOUNT

It contains information about remote devices to be mounted. It is an array of 8 x 3 elements having the following meaning:

[1]: Name of the host of device to be mounted eg.sun01

[2]: Remote device name eg. /c

[3]: Local name of device eg. XP:

12.268 \$NET_Q_STR: Ethernet Remote Interface Information

Memory category static

Load category: not saved

Data type: array of string of two dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is an array of 6x3 elements having the following meaning:

[*,1]: Local IP address of connection

[*,2]: Subnet mask

[*,3]: Remote IP address of connection

With the row indices representing

1 = COM1:

2 = COM2:

3 = COM3:

4 = MDM:

5 = USB Cable

6 = USB Ethernet adaptor

12.269 \$NET_R_STR: Ethernet Remote Interface Setup

Memory category: static
Load category: controller. *Minor category* - environment
Data type: array of string of two dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 6x4 elements having the following meaning:
 [* ,1]: Local IP address of connection
 [* ,2]: Subnet mask
 [* ,3]: Remote IP address of connection
 [* ,4]: Hostname
 With the row indices representing
 1 = COM1:
 2 = COM2:
 3 = COM3:
 4 = MDM:
 5 = USB Cable
 6 = USB Ethernet adaptor

12.270 \$NET_S_INT: Ethernet Network Server Setup

Memory category: static
Load category: controller. *Minor category* - environment
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 3 elements having the following meaning:
 [1]: Number of servers
 [2]: Timeout for a server in minutes
 [3]: Server startup time

12.271 \$NET_T_HOST: Ethernet Network Time Protocol Host

Memory category: static
Load category: controller. *Minor category* - environment
Data type: string
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Host for SNTP protocol or empty if accept incoming time setting

12.272 \$NET_T_INT: Ethernet Network Timer

Memory category: static
Load category: controller. *Minor category* - environment
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 2 elements having the following meaning:
[1]: Timeout of the simple network time protocol
[2]: reserved

12.273 \$NOLOG_ERROR: Exclude messages from logging

Memory category: static
Load category: controller. *Minor category* - shared
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is an array of 8 elements. If an error code is specified in one element of this variable, and this error occurs, the error is not registered in the log files of errors (.LBE). Element 1 and 2 are set by default respectively to 28694 ("Emergency Stop") and 28808 ("Safety gate")

12.274 \$NUM ALOG FILES: Number of action log files

Memory category: static
Load category: controller. *Minor category* - shared
Data type: integer
Attributes: none
Limits: 1..100
Default: 10
S/W Version: 1.0
Description: It represents the number of action log files

12.275 \$NUM ARMS: Number of arms

Memory category: static
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: 1..4
Default: 4
S/W Version: 1.0
Description: It represents the number of arms present in the controller

12.276 \$NUM AUX AXES: Number of auxiliary axes

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the number of auxiliary axes for a particular arm

12.277 \$NUM DBS: Number of allowed Databases

Memory category: static
Load category: controller. *Minor category* - shared
Data type: integer

Attributes: none
Limits 1..30
Default: 10
S/W Version: 1.0
Description: It represents the number of database structures that can be supported by the System.

12.278 \$NUM_DEVICES: Number of Devices

Memory category static
Load category: controller. *Minor category - shared*
Data type: integer
Attributes: none
Limits 4..30
Default: 20
S/W Version: 1.0
Description: It represents the number of devices which can be defined at any one time.

12.279 \$NUM_IO_DEV: Number of Input/Output Devices

Memory category static
Load category: input/output *Minor category - fieldbus*
Data type: integer
Attributes: privileged read-write, property
Limits 1..64
Default: 32
S/W Version: 1.0
Description: It represents the number of Input/Output devices present in the Controller.

12.280 \$NUM_IO_STS: Number of Input/Output points

Memory category static
Load category: input/output *Minor category - fieldbus*
Data type: integer
Attributes: privileged read-write, property
Limits 256..1024
Default: reserved
S/W Version: 1.0
Description: It represents the number of Input/Output points present in the Controller.

12.281 \$NUM_JNT_AXES: Number of joint axes

Memory category field of arm_data
Load category: arm. *Minor category - configuration*
Data type: integer
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the number of joint axes for a particular arm

12.282 \$NUM_LUNS: Number of LUNs

Memory category static
Load category: controller. *Minor category - shared*
Data type: integer
Attributes: none
Limits 1..30
Default: 20
S/W Version: 1.0
Description: It represents the maximum number of LUNs (logical unit numbers) which can be used at any one time (see [OPEN FILE Statement](#)).

12.283 \$NUM_MB: Number of motion buffers

Memory category static
Load category: controller. *Minor category - shared*
Data type: integer
Attributes: none
Limits 5..50
Default: 10
S/W Version: 1.0
Description: It represents the number of concurrent motions which can be in the execution pipeline at the same time.

12.284 \$NUM_MB_AHEAD: Number of motion buffers ahead

Memory category static
Load category: controller. *Minor category - shared*

Data type: integer
Attributes: none
Limits 10
Default: 4
S/W Version: 1.0
Description: It represents the number of motion buffers to look ahead while executing paths. This variable is particularly useful when executing very closed path nodes so to improve the motion execution.

12.285 \$NUM_PROGS: Number of active programs

Memory category static
Load category: not saved
Data type: integer
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: It represents the number of active PDL2 programs at any one time

12.286 \$NUM_PROG_TIMERS: Number of program timers (\$PROG_TIMER_xxx)

Memory category static
Load category: controller *Minor category - environment*
Data type: integer
Attributes: none
Limits 10..50
Default: 16
S/W Version: 1.0
Description: It represents the number of \$PROG_TIMER_O, \$PROG_TIMER_OS, \$PROG_TIMER_X, \$PROG_TIMER_XS timers available at PDL2 program level.

12.287 \$NUM_SCRNS: Number of screens

Memory category static
Load category: controller. *Minor category - environment*
Data type: integer
Attributes: none
Limits 4..12
Default: 9

S/W Version: 1.0

Description: It represents the number of screens available at the PDL2 program level. Screens can be created, deleted, added and removed by the user with the SCRn_CREATE, SCRn_DEL, SCRn_ADD and SCRn_REMOVE built-in routines. This value can be changed by the user, but has effect only if it is saved in the configuration file and the controller is restarted.

12.288 \$NUM_TIMERS: Number of timers

Memory category static

Load category: controller. *Minor category - environment*

Data type: integer

Attributes: none

Limits 10..110

Default: 100

S/W Version: 1.0

Description: It represents the number of \$TIMER and \$TIMER_S available at the PDL2 program level.

12.289 \$NUM_TREES: Number of trees

Memory category static

Load category: controller. *Minor category - shared*

Data type: integer

Attributes: none

Limits 1..110

Default: 10

S/W Version: 1.0

Description: It represents the number of tree structures that can be supported by the System.

12.290 \$NUM_VP2_SCRNS: Number of Visual PDL2 screens

Memory category static

Load category: controller. *Minor category - environment*

Data type: integer

Attributes: none

Limits 4..100

Default: 9

S/W Version: 1.0

Description: It represents the number of Visual PDL2 screens available at the PDL2 program level. Screens can be created, deleted, added and removed by the user with the VP2_SCRN_CREATE Built-In Function, VP2_SCRN_DEL Built-in Procedure, VP2_SCRN_ADD Built-in Procedure and VP2_SCRN_REMOVE Built-in Procedure (for further information see **VP2 - Visual PDL2** Manual, chapter 6). This value can be changed by the user, but it only has effect if it is saved in the configuration file (.C5G) and the Controller is restarted.

12.291 \$NUM_WEAVES: Number of weaves (WEAVE_TBL)

Memory category static
Load category: not saved
Data type: integer
Attributes: read-only
Limits 1..16
Default: 10
S/W Version: 1.0
Description: It represents the number of weave table elements available at the PDL2 program level

12.292 \$ODO_METER: average TCP space

Memory category field of crnt_data
Load category: not saved
Data type: real
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the total arm space. It is only used in cartesian motions for indicating the average TCP space expressed in millimeters.
 This variable is updated every tick (default 2 milliseconds) in case of linear or circular motions.
 It is zeroed in the following situations: restart, reset from pdl2 , when the variable overflows its numerical value

12.293 \$ON_POS_TBL: ON POS table data

Memory category dynamic
Load category: not saved
Data type: array of on_pos_tbl of one dimension
Attributes: none
Limits none

Default:

S/W Version: 1.0

Description: \$ON_POS_TBL is an array of 8 elements, with each schedule containing the \$OP_xxx fields

12.294 \$OP_JNT: On Pos jointpos

Memory category field of on_pos_tbl

Load category: not saved

Data type: jointpos

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: This is the joint position associated to a \$ON_POS_TBL element. As this is a read-only variable, you can assign a value to it by using the ON_JNT_SET built-in routine

12.295 \$OP_JNT_MASK: On Pos Joint Mask

Memory category field of on_pos_tbl

Load category: not saved

Data type: integer

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: This is the mask of joints that are checked when the On Pos feature uses the \$ON_POS_TBL[on_pos_idx].OP_JNT variable. If this mask is 0, \$ON_POS_TBL[on_pos_idx].OP_POS will be used. This mask assumes the value that is passed as parameter to the ON_JNT_SET built-in routine

12.296 \$OP_POS: On Pos position

Memory category field of on_pos_tbl

Load category: not saved

Data type: position

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: This is the position associated to a \$ON_POS_TBL element

12.297 \$OP_REACHED: On Pos position reached flag

Memory category: field of on_pos_tbl

Load category: not saved

Data type: boolean

Attributes: read-only

Limits: none

Default:

S/W Version: 1.0

Description: If \$ON_POS_TBL[on_pos_index].OP_REACHED is TRUE it indicates that the arm is in the sphere region around the \$ON_POS_TBL[on_pos_index].OP_POS position

12.298 \$OP_TOL_DIST: On Pos-Jnt Tolerance distance

Memory category: field of arm_data

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: privileged read-write, property

Limits: none

Default:

S/W Version: 1.0

Description: This variable contains a tolerance in millimeters. The value is related to: X,Y,Z coordinates of the actual Cartesian robot POSITION in respect with the \$ON_POS_TBL[on_pos_idx].OP_POS; linear joints of the actual JOINTPOS in respect with the \$ON_POS_TBL[on_pos_idx].OP_JNT. The ON POS feature must be enabled (ON_POS(ON...)) after having been defined on the position (ON_POS_SET) or on the jointpos (ON_JNT_SET)

12.299 \$OP_TOL_ORNT: On Pos-Jnt Tolerance Orientation

Memory category: field of arm_data

Load category: arm. *Minor category* - configuration

Data type: real

Attributes: privileged read-write, property

Limits: none

Default:

S/W Version: 1.0

Description: This variable contains a tolerance in degrees. The value is related to: the three Euler

angles of the actual Cartesian robot POSITION in respect with the \$ON_POS_TBL[on_pos_idx].OP_POS; rotating joints of the actual JOINTPOS in respect with the \$ON_POS_TBL[on_pos_idx].OP_JNT. The ON POS feature must be enabled (ON_POS(ON,...)) after having been defined on the position (ON_POS_SET) or on the jointpos (ON_JNT_SET)

12.300 \$OP_TOOL: The On Pos Tool

<i>Memory category:</i>	field of on_pos_tbl
<i>Load category:</i>	not saved
<i>Data type:</i>	position
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable is the tool associated to \$ON_POS_TBL[on_pos_idx].OP_POS. It must always be initialized unless \$ON_POS_TBL[on_pos_idx].OP_TOOL_DSBL is set to TRUE

12.301 \$OP_TOOL_DSBL: On Pos tool disable flag

<i>Memory category:</i>	field of on_pos_tbl
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	If this variable is TRUE, the tool is not considered for validating the reaching of the \$ON_POS_TBL[on_pos_idx].OP_POS position

12.302 \$OP_TOOL_RMT: On Pos Remote tool flag

<i>Memory category:</i>	field of on_pos_tbl
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	If TRUE, the remote tool is enabled for the corresponding \$ON_POS_TBL element. In this case the \$ON_POS_TBL[on_pos_idx].OP_TOOL_DSBL should be set to FALSE and \$ON_POS_TBL[on_pos_idx].OP_POS, \$ON_POS_TBL[on_pos_idx].OP_TOOL and

\$ON_POS_TBL[on_pos_idx].OP_UFRAME must all have been initialised

12.303 \$OP_UFRAME: The On Pos Uframe

Memory category field of on_pos_tbl

Load category: not saved

Data type: position

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: This is the User frame associated to a certain \$ON_POS_TBL element

12.304 \$ORNT_TYPE: Type of orientation

Memory category program stack

Load category: not saved

Data type: integer

Attributes: WITH MOVE; MOVE ALONG

Limits EUL_WORLD..RS_TRAJ

Default: RS_WORLD

S/W Version: 1.0

Description: It represents the type of evolution to be used for motion orientation. Valid values are represented by the predefined constants EUL_WORLD (3 angle), RS_WORLD (2 angle relative to world), RS_TRAJ (2 angle relative to trajectory) and WRIST_JNT (wrist).

12.305 \$OT_COARSE: On Trajectory indicator

Memory category field of crnt_data

Load category: not saved

Data type: boolean

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: This boolean variable indicates if the TCP (tool center point) is on the trajectory (TRUE) or not (FALSE)

12.306 \$OT_JNT: On Trajectory joint position

Memory category field of crnt_data

Load category: not saved
Data type: jointpos
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is the current position of robot joints (JOINTPOS) on the trajectory. Its value is updated any time the robot stops

12.307 \$OT_POS: On Trajectory position

Memory category field of crnt_data
Load category: not saved
Data type: position
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is the current robot position (POSITION) on the trajectory. Its value is updated any time the robot stops

12.308 \$OT_TOL_DIST: On Trajectory Tolerance distance

Memory category field of arm_data
Load category: arm. *Minor category - configuration*
Data type: real
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This variable contains the tolerance (in millimeters related to the X,Y,Z coordinates) of the current robot position in respect with \$CRNT_DATA[arm].OT_POS. It is also used for expressing the tolerance related to linear auxiliary axes

12.309 \$OT_TOL_ORNT: On Trajectory Orientation

Memory category field of arm_data
Load category: arm. *Minor category - configuration*
Data type: real
Attributes: privileged read-write

Limits none
Default: 3
S/W Version: 1.0
Description: This variable contains the tolerance (in degrees related to the Euler angles) of the current robot position in respect with \$CRNT_DATA[arm].OT_POS. It is also used for expressing the tolerance related to rotating auxiliary axes

12.310 \$OT_TOOL: On Trajectory TOOL position

Memory category field of crnt_data
Load category: not saved
Data type: position
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is the Tool frame related to the \$CRNT_DATA[arm_num].OT_POS position

12.311 \$OT_TOOL_RMT: On Trajectory remote tool flag

Memory category field of crnt_data
Load category: not saved
Data type: boolean
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: If this variable is TRUE, the remote tool is enabled for the \$CRNT_DATA[arm_num].OT_POS variable

12.312 \$OT_UFRAME: On Trajectory User frame

Memory category field of crnt_data
Load category: not saved
Data type: position
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This is the User frame related to the \$CRNT_DATA[arm_num].OT_POS position

12.313 \$OT_UNINIT: On Trajectory position uninit flag

Memory category: field of crnt_data
Load category: not saved
Data type: boolean
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: If TRUE it means that the \$CRNT_DATA[arm_num].OT_POS variable is not initialised

12.314 \$OUT: OUT digital

Memory category: port
Load category: not saved
Data type: array of boolean of one dimension
Attributes: privileged read-write, pulse usable
Limits: none
Default:
S/W Version: 1.0
Description: It represents the set of digital output points reserved for applications.
 For further information see also [par. 5.2.2 \\$IN and \\$OUT on page 104](#).

12.315 \$PAR: Nodal motion variable



Note that in this section (“\$PAR: Nodal motion variable”), any occurrence of the word ‘nodal’ always refers to the MOVE WITH \$PAR nodal approach.

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: WITH MOVE
Limits: 7..10
Default: 7
S/W Version: 1.0

Description:

This variable can be used in the WITH clause of a [MOVE Statement](#) for implementing one nodal modality for MOVE. It consists in assigning the INTEGER result of a user-written function call to the \$PAR variable. The parameters to this function should be the values to assign to those predefined motion variables to be applied to the move. If the parameters are function calls themselves, they should be as short as possible for avoiding slowing down the execution of each move.

Example:

routine that calculates the tool and the frame, exported from pex5 program

```

ROUTINE tl_fr(ai_tool_idx,ai_frame_idx: INTEGER): INTEGER
EXPORTED FROM pex5

ROUTINE velo(ai_spd: INTEGER): INTEGER
BEGIN
    $ARM_SPD_OVR := ai_spd
    RETURN(0)
END velo
ROUTINE ter(ai_term: INTEGER): INTEGER
BEGIN
    $TERM_TYPE := ai_term
    RETURN(0)
END ter
ROUTINE func(par1,par2,par3: INTEGER): INTEGER
BEGIN
    RETURN(0)
END func
MOVE JOINT TO jnt0001p WITH $PAR = func(tl_fr (1,3), velo(20),
ter(NOSETTLE))
MOVE JOINT TO jnt0002p WITH $PAR = func(tl_fr (1,3),
velo(20),ter(NOSETTLE))
MOVE LINEAR TO pnt0001p WITH $PAR = func(tl_fr (1,2),
velo(40), ter(COARSE))

```

Here follow some suggestions to be followed when using \$PAR:

With the \$PAR it is suggested not to use the following names as they are reserved for a specific application: tf, rtf, v, vl, zone, gp, mp, ap, orn.

The function WITHed with the \$PAR (function func in the above example) should always return zero.

If the \$PAR is adopted for the [MOVE Statements](#), it is a good rule of programming to use it in each move. A mixed way of programming (nodal and modal) is not recommended.

If the [MOVE Statement](#) includes multiple WITH clauses (not recommended way of programming), the WITH \$PAR should be the first one, otherwise the other WITH clauses would have no effect.

The [MOVE Statement](#) should always have the trajectory specified (LINEAR, JOINT, CIRCULAR)

12.316 \$PGOV_ACCURACY: required accuracy in cartesian motions

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: real
Attributes: WITH MOVE, MOVE ALONG
Limits 0.2 / 30.0
Default: 2.580
S/W Version: 1.0
Description: it represents the accuracy for linear and circular cartesian move, as maximum cartesian error desired (in mm). The value to assign to this variable depends on the level of accuracy that the user wants to reach during the requested path. Please note that , if \$ARM_DATA[<arm_num>].PGOV_SPD_REDUCTION has the value of 0, this variable is ignored by the motion environment

12.317 \$PGOV_MAX_SPD_REDUCTION: Maximum speed scale factor

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: integer
Attributes: WITH MOVE, MOVE ALONG
Limits 0 .. 95
Default: 30
S/W Version: 1.0
Description: it represents the maximum percentage of loss in cartesian speed that is allowed in order to improve the cartesian accuracy. If set to 0, no increase in accuracy will be applied. If it is not correctly set, according to the maximum linear speed required by the work cycle, the accuracy (set in \$PGOV_ACCURACY predefined variable) might not be reached and also the orientation error, if it occurs, would not be compensated.

12.318 \$PGOV_ORNT_PER: percentage of orientation

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: integer
Attributes: WITH MOVE, MOVE ALONG
Limits 0..100
Default: 0
S/W Version: 1.0

Description: it represents the percentage of error, in orientation, which need to be compensated. This percentage should be increased only in case the orientation error is too high in respect to what expected with the Path Governor enabled. The default value is 0.

12.319 \$POS_LIMIT_AREA: Cartesian limits of work area

Memory category field of arm_data

Load category: arm. *Minor category* - configuration

Data type: array of real of one dimension

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: This array represents the cartesian limits of the work area

12.320 \$PREG: Position registers - saved

Memory category static

Load category: controller. *Minor category* - vars

Data type: array of position of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Position registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Jointpos and Xtndpos datatypes there are saved and non-saved registers.

12.321 \$PREG_NS: Position registers - not saved

Memory category static

Load category: not saved

Data type: array of position of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Non-saved Position registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Jointpos and Xtndpos datatypes there are saved and non-saved registers.

12.322 \$PROG_ACC_OVR: Program acceleration override

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE
Limits 1..100
Default: 100
S/W Version: 1.0
Description: It represents the acceleration override in percentage for motions issued from this program. Changing its value does not affect deceleration or speed. It is only used before the beginning of the motion, during the motion planning phase. When this variable is modified it does not affect the active motion, but will be used in any new motions.

12.323 \$PROG_ARG: Program's activation argument

Memory category: program stack
Load category: not saved
Data type: STRING
Attributes: none
Limits none
Default: ""
S/W Version: 1.0
Description: It contains the line argument passed in when the program was activated.

Example: the listed below program displays the argument which is passed while activating 'showarg1'.

```

PROGRAM showarg1 NOHOLD
BEGIN
    WRITE LUN_CRT('The argument is: ', $PROG_ARG, NL) -- display
                                         -- the passed
                                         -- argument
END showarg1

```

12.324 \$PROG_ARM: Arm of program

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: read-only
Limits 1..4

Default:

S/W Version: 1.0

Description: It represents the current arm for this program. Statements and expressions that need arm numbers use this value if the arm number is not specified. If the program attribute \$PROG_ARM is not specified, the value of \$DFT_ARM is used.

12.325 \$PROG_ARM_DATA: Arm-related Data - Program specific

Memory category program stack

Load category: not saved

Data type: array of prog_arm_data of one dimension

Attributes: none

Limits none

Default: none

S/W Version: 1.15

Description: \$PROG_ARM_DATA is an array of predefined records with one element for each arm and local to a program. The fields of each element represent arm-related data. It is not always necessary to specify the \$PROG_ARM_DATA prefix when referring to a field of \$PROG_ARM_DATA.

The field of \$PROG_ARM_DATA that is used by default is the one for the arm specified in PROG_ARM.

12.326 \$PROG_BREG: Boolean Registers - Program specific

Memory category program stack

Load category: not saved

Data type: array of boolean of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Boolean registers are “program specific” variables that can be used by users instead of having to define variables.

12.327 \$PROG_CNG: Program configuration

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>Certain aspects of the execution of a program can be configured using this predefined variable. The bits are described below.</p> <ul style="list-style-type: none"> – Bit 1: The robot will not move at a reduced speed (for safety) after the program execution has been interrupted. – Bit 2: The robot will not move at reduced speed (for safety) when the first motion of the program is executed. – Bit 3: The program cannot set an output when in particular circumstances: <ul style="list-style-type: none"> • if in PROGR state from an active program. • b: if in PROGR state executing the statement from the WINC5G program running on the PC when the teach pendant is recognised to be out of cabinet.c: if in PROGR state under MEMORY DEBUG or PROGRAM EDIT from the WINC5G program running on the PC when the teach pendant is recognised to be out of cabinet. • d: if the state selector was turned out of the T1 position when the teach pendant is recognised to be out of cabinet. This bit is copied from bit 3 of \$CNTRL_CNFG when program is activated. – Bit 4: The program cannot be deactivated from the command menu. Note that if the deactivation is issued from a programming environment or EXECUTE, the program will still be deactivated. Note also that if the program is holdable and the first START after the activation has not been pressed yet, the program can still be deactivated. – Bit 5: If set, the WAIT FOR does not trigger if the program is held – Bit 6: If set, upon a SYS_CALL error the program is not paused in a similar way as when issuing an ERR_TRAP_ON (39960). Note that error 39960 is not trapped on also if this bit has value 1. – Bit 7-24: reserved – Bit 25: Set to 1 if the UNICODE characters coding is handled by the system – Bit 26-32: reserved

12.328 \$PROG_CONDS: Defined conditions of a program

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents the condition handlers currently defined in the program. Each element of this variable should be read as a mask of bits, in which each bit corresponds to a condition handler number.</p> <p>For each element of this variable, only bits 1-30 are significative (bits 31 and 32 should be ignored).</p> <p>Here follows an example of how this variable should be read for understanding which condition is defined for a certain program.</p>

Assume that the program has condition handlers 1, 30, 31, and 32 defined: \$PROG_CONDS[1] will be 0x20000001 (bits 1 and 30 set to 1), \$PROG_CONDS[2] will be 0x3 (bits 1 and 2 set), and the remaining elements will be 0.

12.329 \$PROG_DEC_OVR: Program deceleration override

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	WITH MOVE
<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the deceleration override percentage for motions issued from this program. Changing its value does not affect acceleration or speed. It is only used before the motion begins, during the motion planning phase. When this variable is modified it does not influence an active motion, but will be used for any new motions.

12.330 \$PROG_IREG: Integer Registers - Program specific

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	Integer registers are “program specific” variables that can be used by users instead of having to define variables.

12.331 \$PROG_LINE: executing program line

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0

Description: It represents the line number currently being executed

12.332 \$PROG_LUN: program specific default LUN

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It represents the default LUN for the current program. When the program is activated, this is initialized with \$DFT_LUN.

12.333 \$PROG_NAME: Executing program name

Memory category program stack

Load category: not saved

Data type: string

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: It represents the name of the executing program. This variable is especially useful for determining, within the context of a library routine, which is the program that calls that routine in order to undertake different actions basing on it. Note that the program name is stored in upper case.

12.334 \$PROG_OWNER: Program Owner of executing line

Memory category program stack

Load category: not saved

Data type: string

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: It represents the program owner for the line number currently being executed.

12.335 \$PROG_RES: Program's execution result

Memory category: program stack
Load category: not saved
Data type: string
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It contains the result from the execution of a routine.

12.336 \$PROG_RREG: Real Registers - Program specific

Memory category: program stack
Load category: not saved
Data type: array of real of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Real registers are “program specific” variables that can be used by users instead of having to define variables.

12.337 \$PROG_SPD_OVR: Program speed override

Memory category: program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE
Limits: 1..100
Default: 100
S/W Version: 1.0
Description: It represents the speed override in percentage for motions issued from this program. Changing this value does not affect acceleration or deceleration

12.338 \$PROG_SREG: String Registers - Program specific

Memory category: program stack

Load category: not saved
Data type: array of string of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: String registers are “program specific” variables that can be used by users instead of having to define variables.

12.339 \$PROG_TIMER_O: Program timer - owning context specific (in ms)

Memory category port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents 1 millisecond timers. There are \$NUM_PROG_TIMERS. Such timers are specific per program owning context.
See also [\\$PROG_TIMER_X: Program timer - execution context specific \(in ms\)](#),
[\\$PROG_TIMER_OS: Program timer - owning context specific \(in seconds\)](#),
[\\$PROG_TIMER_XS: Program timer - execution context specific \(in seconds\)](#), [\\$TIMER: Clock timer \(in ms\)](#), [\\$TIMER_S: Clock timer \(in seconds\)](#).

12.340 \$PROG_TIMER_OS: Program timer - owning context specific (in seconds)

Memory category port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents second timers. There are \$NUM_PROG_TIMERS. Such timers are specific per program owning context. See also:
[\\$PROG_TIMER_O: Program timer - owning context specific \(in ms\)](#), [\\$PROG_TIMER_X: Program timer - execution context specific \(in ms\)](#), [\\$PROG_TIMER_XS: Program timer - execution context specific \(in seconds\)](#), [\\$TIMER: Clock timer \(in ms\)](#), [\\$TIMER_S: Clock timer \(in seconds\)](#).

12.341 \$PROG_TIMER_X: Program timer - execution context specific (in ms)

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents 1 millisecond timers. There are \$NUM_PROG_TIMERS. Such timers are specific per program executing context. [\\$PROG_TIMER_O: Program timer - owning context specific \(in ms\)](#), [\\$PROG_TIMER_OS: Program timer - owning context specific \(in seconds\)](#), [\\$PROG_TIMER_XS: Program timer - execution context specific \(in seconds\)](#), [\\$TIMER: Clock timer \(in ms\)](#), [\\$TIMER_S: Clock timer \(in seconds\)](#).

12.342 \$PROG_TIMER_XS: Program timer - execution context specific (in seconds)

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents second timers. There are \$NUM_PROG_TIMERS. Such timers are specific per program execution context. See also [\\$PROG_TIMER_X: Program timer - execution context specific \(in ms\)](#), [\\$PROG_TIMER_O: Program timer - owning context specific \(in ms\)](#), [\\$PROG_TIMER_OS: Program timer - owning context specific \(in seconds\)](#), [\\$TIMER: Clock timer \(in ms\)](#), [\\$TIMER_S: Clock timer \(in seconds\)](#).

12.343 \$PROP_AUTHOR: last Author who saved the file

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of string
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0

Description: Username of the last person who saved the file.

Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.344 \$PROP_DATE: date and time when the program was last saved

Memory category program loaded

Load category: when program loaded into memory

Data type: array of string

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: Date when the program was last saved.

Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.345 \$PROP_FILE: property information for loaded file

Memory category program loaded

Load category: when program loaded into memory

Data type: array[5] of string

Attributes: read-only, property

Limits none

Default:

S/W Version: 1.0

Description: Represents the last loaded file

- [1]: COD file,
- [2]: VAR file,
- [3]: VPS file,
- [4]: VTR file,
- [5]: CIO file.

12.346 \$PROP_HELP: the help the user wants

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of string
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: Represents the property containing the help for the program. Each string is 127 characters.
Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.347 \$PROP_HOST: Controller ID or PC user domain, upon which the file was last saved

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of string
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: Represents the host upon which the file was generated.
Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.348 \$PROP_REVISION: user defined string representing the version

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of string
Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: Represents the revision note (max length 15).

Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.349 \$PROP_TITLE: the title defined by the user for the file

Memory category program loaded

Load category: when program loaded into memory

Data type: array of string

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: Represents the property containing the title for the program. Each string is 30 characters.

Elements:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.350 \$PROP_UML: user modify level

Memory category program loaded

Load category: when program loaded into memory

Data type: array of integer of one dimension

Attributes: privileged read-write, property

Limits 0..7

Default:

S/W Version: 1.0

Description: Represents the the property help:

- [1] for .COD files,
- [2] for .VAR files,
- [3] for .VPS files,
- [4] for .VTR files,
- [5] for .CIO files.

12.351 \$PROP_UVL: user view level

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of integer of one dimension
Attributes: privileged read-write, property
Limits: 0..7
Default:
S/W Version: 1.0
Description: Represents the the property help:
– [1] for .COD files,
– [2] for .VAR files,
– [3] for .VPS files,
– [4] for .VTR files,
– [5] for .CIO files.

12.352 \$PROP_VERSION: version upon which it was built

Memory category: program loaded
Load category: when program loaded into memory
Data type: array of string
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: Represents the version when the file was generated.
Elements:
– [1] for .COD files,
– [2] for .VAR files,
– [3] for .VPS files,
– [4] for .VTR files,
– [5] for .CIO files.

12.353 \$PWR_RCVR: Power failure recovery mode

Memory category: static
Load category: controller. *Minor category* - shared
Data type: integer
Attributes: none
Limits: 0..1
Default: 1
S/W Version: 1.0

Description: It represents the mode for power failure recovery. A value of 1 means that the controller will recover to the same state as it was before the power failure. A value of zero means that the controller will perform a true restart.

12.354 \$RAD_IDL_QUO: Radian ideal quote

Memory category field of crnt_data

Load category: not saved

Data type: array of real of one dimension

Attributes: read-only

Limits: none

Default:

S/W Version: 1.0

Description: It represents the ideal position in radians or millimeters. It does not take into consideration the coupling effect between axes

12.355 \$RAD_OVR: TP Rotational Acc/Dec Override for FLY CART

Memory category field of crnt_data

Load category: not saved

Data type: integer

Attributes: none

Limits: 1..100

Default: 100

S/W Version: 1.0

Description: It is possible to dynamically decrease the limit of the rotational acceleration deceleration parameters.

This override mainly modulates the rotational acceleration and deceleration. It changes [\\$LIN_ACC_LIM: Linear acceleration limit](#) and [\\$LIN_DEC_LIM: Linear deceleration limit](#) with the value of its percentage. It works only for the FLY CART Algorithm.

12.356 \$RAD_TARG: Radian target

Memory category field of crnt_data

Load category: not saved

Data type: array of real of one dimension

Attributes: read-only

Limits: none

Default:

S/W Version: 1.0

Description: It represents the desired arm position expressed in radians or millimeters

12.357 \$RAD_VEL: Radian velocity

Memory category: field of crnt_data
Load category: not saved
Data type: array of real of one dimension
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: It represents the current axis velocity expressed in radians per tick or millimeters per tick

12.358 \$RBT_CNFG: Robot board configuration

Memory category: static
Load category: not saved
Data type: integer
Attributes: read-only
Limits: none
Default:
S/W Version: 1.0
Description: It represents the different robot configuration and options present in the system:

- Bit 1..8: reserved
- Bit 9..12: Loaded language:
 - 0: English
 - 1: Italian
 - 2: French
 - 3: German
 - 4: Spanish
 - 5: Portuguese
 - 6: Turkish
 - 7: Chinese
 - 8: Czech
- Bit 13..16: reserved
- Bit 17: System variables initialized
- Bit 18: XD: device connected
- Bit 19: TX: device connected
- Bit 20..21: reserved
- Bit 22: USB NET (ethernet) device inserted
- Bit 23..27: reserved
- Bit 28: used for language localization; if the set language is Portuguese this bit indicates whether the language is for Portugal (value 0) or for Brazil (value 1); if the set language is English, this bit indicates whether the language is for United Kingdom (value 0) or for USA (value 1)
- Bit 29..30: reserved
- Bit 31: system in minimal configuration
- Bit 32: reserved

12.359 \$RB_FAMILY: Family of the robot arm

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the family of the robot arm. The different families are:
 – 1: SMART
 – 2: MAST
 – 3: reserved
 – 4: C5G (without inverse and direct kinematics)
 – 5: PMAST
 – 6..18: reserved

12.360 \$RB_MODEL: Model of the robot arm

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the model of the robot arm. A different number is used to represent each different model of robot

12.361 \$RB_NAME: Name of the robot arm

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: string
Attributes: privileged read-write, property
Limits: none
Default:
S/W Version: 1.0
Description: It represents the COMAU name of the robot arm

12.362 \$RB_STATE: State of the robot arm

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the current state of the robot arm. Some states have been defined.
 – Bit 1: The arm is not RESUMED
 – Bit 2: The arm is not CALIBRATED
 – Bit 3: The arm is in HELD state
 – Bit 4: The arm is in DRIVE OFF state
 – Bit 5: The arm is in SIMULATE state

12.363 \$RB_VARIANT: Variant of the robot arm

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: privileged read-write,property
Limits: none
Default:
S/W Version: 1.0
Description: It is a bit mask used to configure specific features of some robot models. The meaning of the bits is internal

12.364 \$RCVR_DIST: Distance from the recovery position

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: It represents the distance, expressed in millimeters, covered moving backward after having recovered the interrupted trajectory ("process resume" modality). It has effect on either joint and Cartesian trajectories. The backward phase begins at the START after every stop command (HOLD, LOCK, Emergency stop, Power failure). The backward can

involve also the previous movement (with respect to the current) on condition that it was in fly. It is enabled only in AUTO state and when \$RCVR_TYPE is set to the value 0 or 4.

12.365 \$RCVR_LOCK: Change arm state after recovery

Memory category: field of crnt_data

Load category: not saved

Data type: boolean

Attributes: none

Limits: none

Default:

S/W Version: 1.0

Description: If this variable is set to TRUE, the arm is automatically set in a resumable state after a recovery motion. In order to move again that arm, a [RESUME Statement](#) must be issued, either from within a program or using the Execute command present on the system menu

12.366 \$RCVR_TYPE: Type of motion recovery

Memory category: field of arm_data

Load category: arm. *Minor category* - configuration

Data type: integer

Attributes: property

Limits: 0..9

Default: 0

S/W Version: 1.0

Description: It represents the type of motion recovery after a motion has been interrupted. The possible values and their recovery types actions are listed below.

- 0: Recovery to the interrupted trajectory with a joint interpolation.
- 1: Recovery to the initial position of the interrupted motion with a joint interpolation.
- 2: Recovery to the final position of the interrupted motion with a joint interpolation.
- 3: Do not recover; an error message is returned to the user.
- 4: Recovery on the interrupted trajectory, with the same motion type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.
- 5: Recovery on the initial position of the interrupted trajectory, with the same motion type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.
- 6: Recovery on the final position of the interrupted trajectory, with the same motion type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.
- 7: Reserved

- 8: Reserved.
- 9: automatic recovery of the process. The functionality is the same as modality 4 but the distance undertaken in the movement of return is the result of the sum of **\$RCVR_DIST: Distance from the recovery position** value and the covered distance in manual motion after the robot stopping on the planned movement.

12.367 \$READ_TOUT: Timeout on a READ

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	0..2147483647
<i>Default:</i>	0
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the timeout in milliseconds for a READ. The default value is zero, meaning that the READ will not timeout.

12.368 \$REC_SETUP: RECord key setup

<i>Memory category</i>	static
<i>Load category:</i>	settings
<i>Data type:</i>	string
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	reserved
<i>S/W Version:</i>	1.0
<i>Description:</i>	It contains information on the current setup of the REC key. This information can only be changed via the IDE Page, Select (F4) menu, REC setup function. For further information see Control Unit Use manual - IDE Page paragraph

12.369 \$REMOTE: Functionality of the key in remote

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category - shared</i>
<i>Data type:</i>	boolean
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	TRUE/FALSE
<i>Default:</i>	TRUE
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable can be used for entering the LOCAL state with the state selector is in the REMOTE position. This can be achieved setting \$REMOTE to FALSE, allowing the program execution in LOCAL at the maximum speed.

12.370 \$REM_I_STR: Remote connections Information

Memory category: static
Load category: not saved
Data type: array of string of two dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It is a 2 dimensional array with each element containing the information of who is connected to the controller. The first dimension can assume: value 1 for the host, value 2 for the user. The second dimension identifies the program used for connecting to the controller: value 1 is for WinC5G, value 3 is for the TP, and other values are used for external connections. Example: \$REM_I_STR[1,1] shows the computer that is connected via WinC5G to the controller. \$REM_I_STR[2,1] shows the user that is connected via WinC5G to the controller

12.371 \$REM_TUNE: Internal remote connection tuning parameters

Memory category: static
Load category: controller. *Minor category - shared*
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: This variable is used for setting internal parameters for the communication between the teach pendant and the controller

12.372 \$RESTART: Restart Program

Memory category: static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: it represents the program to be loaded and activated upon the issuing of the restart of the controller

12.373 \$RESTART_MODE: Restart mode

Memory category static
Load category: controller. *Minor category - environment*
Data type: integer
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: During the Restart operation (Cold, reload or shutdown) this value is set to the type of restart. A program can abort the restart by setting its \$PROG_RETURN variable to -1 or an error status.

12.374 \$RESTORE_SET: Default devices

Memory category static
Load category: controller. *Minor category - environment*
Data type: array of string of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: Array of 8 elements with each element representing the definition of a restore Saveset. Savesets are used by FilerUtilityRestore command, when /S option is specified, for specifying where the files should be copied from. For example: if \$RESTORE_SET[2] is set to

```
'pippo4|UD:\dir1\*.pd1'
```

 and the command Filer Utility Restore/Saveset is issued with the **pippo4** parameter as the Saveset name, all the **.pd1** files previously backed up from **UD:\dir1** will be restored back in that directory.
 As a default, the first element (i.e. \$RESTORE_SET[1]) contains the following value

```
'All|TTS_USER_DEV"\*.*\s/x*.lbe/x*.lba'
```

 which means 'All files except error & action logs'. Any other element is set to nilThe following switches are available to properly setup the Saveset:

- **/A** (=After) restores files created/modifed AFTER the specified date
- **/N** (=Newer) restores the most recent files compared with the ones of the .LST file
- **/O** (=Overwrite) overwrites read-only files too
- **/S** (=Subdirectories) include subdirectories
- **/X** (=eXclude) exclude certain files from the restore operation

 The following predefined Saveset is available:

- \$RESTORE_SET[1]: "All|TTS_USER_DEV"*.*\s/x*.lbe/x*.lba"

12.375 \$ROT_ACC_LIM: Rotational acceleration limit

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the maximum rotational acceleration expressed in radians per second squared.

12.376 \$ROT_DEC_LIM: Rotational deceleration limit

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the maximum rotational deceleration expressed in radians per second squared.

12.377 \$ROT_SPD: Rotational speed

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: field node, WITH MOVE; MOVE ALONG
Limits: none
Default:
S/W Version: 1.0
Description: It represents the rotational speed expressed in radians per second. The value is used by the control during the planning phase of the trajectory when [\\$SPD_OPT: Type of speed control](#) has a value different from SPD_JNT, SPD_LIN, or SPD_CONST. In this case the TCP rotational speed corresponds to \$ROT_SPD less a general override percentage, as long as the value is less than [\\$ROT_SPD_LIM: Rotational speed limit](#). In addition, if the auxiliary axis is linear, \$ROT_SPD is not considered no matter if [\\$SPD_OPT: Type of speed control](#) is set to SPD_AUX1 or SPD_AUX2 or SPD_AUX3 or SPD_AUX4.

12.378 \$ROT_SPD_LIM: Rotational speed limit

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: It represents the maximum rotational speed expressed in radians per second.

12.379 \$RREG: Real registers - saved

Memory category: static
Load category: controller. *Minor category* - vars
Data type: array of real of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Real registers that can be used by users instead of having to define variables. Also for Integer, String, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.380 \$RREG_NS: Real registers - not saved

Memory category: static
Load category: not saved
Data type: array of real of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: Non-saved Real registers that can be used by users instead of having to define variables. Also for Integer, String, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.381 \$SAFE_ENBL: Safe speed enabled

Memory category: field of crnt_data
Load category: not saved

<i>Data type:</i>	boolean
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the fact that the current motion is executing at a reduced speed for safety reasons. This flag is set upon the first motion of an activated program or if the user interrupted the program execution flow in a programming environment

12.382 \$SAFE_SPD: User Velocity for testing safety speed

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - configuration</i>
<i>Data type:</i>	real
<i>Attributes:</i>	property
<i>Limits</i>	none
<i>Default:</i>	0.250
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the user velocity, expressed in meters per second, for testing the safety speed.

12.383 \$SDI: System digital input

<i>Memory category</i>	port
<i>Load category:</i>	not saved
<i>Data type:</i>	array of boolean of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the system digital input points. For further information see also par. 5.3.1 \$SDI and \$SDO on page 105 .

12.384 \$SDO: System digital output

<i>Memory category</i>	port
<i>Load category:</i>	not saved
<i>Data type:</i>	array of boolean of one dimension
<i>Attributes:</i>	pulse usable
<i>Limits</i>	none

Default:

S/W Version: 1.0

Description: It represents the system digital output points. For further information see also [par. 5.3.1 \\$SDI and \\$SDO on page 105](#).

12.385 \$SEG_ADV: PATH segment advance

Memory category static

Load category: not saved

Data type: integer

Attributes: limited access, field node, WITH MOVE ALONG

Limits none

Default:

S/W Version: 1.0

Description: It represents whether on MV nodal motion the motion is Advanced or not.

12.386 \$SEG_COND: PATH segment condition

Memory category static

Load category: not saved

Data type: integer

Attributes: limited access, field node, WITH MOVE ALONG

Limits none

Default:

S/W Version: 1.0

Description: Used for setting a CONDITION with a nodal data for use on the MV statement (nodal Move)

12.387 \$SEG_DATA: PATH segment data

Memory category static

Load category: not saved

Data type: boolean

Attributes: limited access, field node, WITH MOVE ALONG

Limits none

Default:

S/W Version: 1.0

Description: If this standard field is included in the node definition (NODEDEF) and the value is set to TRUE for a particular node, the segment data for the last node of the MOVE ALONG is used instead of the program's current data. The default is to use the program's current data (i.e. use \$TERM_TYPE instead of \$SEG_TERM_TYPE). This field is useful when the node data defined in MEMORY TEACH is to be used in a [MOVE ALONG Statement](#)

instead of the program data.

12.388 \$SEG_FLY: PATH segment fly or not

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	boolean
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents whether the motion between the nodes is "flied". The variable has the same meaning as the FLY in a MOVEFLY statement. This standard node field does not apply to the last node, in which case the MOVE or MOVEFLY statement is used to determine whether the motion is flied

12.389 \$SEG_FLY_DIST: Parameter in segment fly motion

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	real
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable has the same purpose as \$FLY_DIST but is used with a PATH segment. \$SEG_FLY_DIST does not apply to the last node of the path, \$FLY_DIST is used instead

12.390 \$SEG_FLY_PER: PATH segment fly percentage

<i>Memory category:</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0
<i>Description:</i>	This represents the fly percentage for a PATH segment. This standard node field has the same meaning as \$FLY_PER. \$SEG_FLY_PER does not apply to the last node, in which case \$FLY_PER is used. If the standard field is not specified in the node definition or in a

WITH clause, the current program value is used.

12.391 \$SEG_FLY_TRAJ: Type of fly control

<i>Memory category</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	FLY_AUTO..FLY_TOL
<i>Default:</i>	FLY_TOL
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable has the same purpose as \$FLY_TRAJ but is applied to a PATH segment. \$SEG_FLY_TRAJ does not apply to the last node of a path, instead the value of \$FLY_TRAJ is used

12.392 \$SEG_FLY_TYPE: PATH segment fly type

<i>Memory category</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	FLY_NORM..FLY_CART
<i>Default:</i>	FLY_NORM
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the fly type for a PATH segment. This standard node field has the same meaning as \$FLY_TYPE. \$SEG_FLY_TYPE does not apply to the last node, in which case \$FLY_TYPE is used. If the standard field is not specified in the node definition (NODEDEF) or in a WITH clause, the current program value is used

12.393 \$SEG_OVR: PATH segment override

<i>Memory category</i>	static
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	limited access, field node, WITH MOVE ALONG
<i>Limits</i>	1..100
<i>Default:</i>	100
<i>S/W Version:</i>	1.0

Description: This represents the acceleration, deceleration, and speed override for a particular PATH segment in a similar way as the [\\$PROG_ACC_OVR: Program acceleration override](#), [\\$PROG_DEC_OVR: Program deceleration override](#), and [\\$PROG_SPD_OVR: Program speed override](#) variables are applied to a regular non-PATH motion. If the standard field is not specified in the node definition (NODEDEF) or in a WITH clause, the current program values are used

12.394 \$SEG_REF_IDX: PATH segment reference index

Memory category static

Load category: not saved

Data type: integer

Attributes: limited access, field node, WITH MOVE ALONG

Limits 0..7

Default: 0

S/W Version: 1.0

Description: Each PATH contains a frame table (FRM_TBL) of 7 positional elements that can be used either as a tool or a reference frame in a node which contains the MAIN_POS standard field. \$SEG_REF_IDX represents the FRM_TBL index of the reference frame to be used by a particular node; if \$SEG_REF_IDX is not included in the node definition or has value of zero (default value), no reference frame is applied to the path segment. As there is just one FRM_TBL in each path, it is suggested to use elements 1 to 3 inclusive for reference frames and elements 4 to 7 for tools.

```

TYPE frm = NODEDEF
    $MOVE_TYPE
    $MAIN_POS
    $SEG_REF_IDX
    $SEG_TOOL_IDX
ENDNODEDEF
-- The nodes of this path should either be taught or NODE_APPended
VAR frmpth: PATH OF frm
    .
    .
    .
BEGIN
frmpth.FRML_TBL[1] := POS(1000, 200, 1000, 0, 180, 0, '')
frmpth.FRML_TBL[4] := POS(100, -200, 0, 0, -90, 0, '')

frmpth.NODE[5].$MOVE_TYPE := LINEAR
frmpth.NODE[5].$MAIN_POS := POS(100, -200, 300, 0, 180, 0, '')
frmpth.NODE[5].$SEG_REF_IDX := 1

frmpth.NODE[6].$MOVE_TYPE := LINEAR
frmpth.NODE[6].$MAIN_POS := POS(-1000, -1000, 1000, 0, 180, 0, '')
frmpth.NODE[6].$SEG_TOOL_IDX := 4

```

12.395 \$SEG_STRESS_PER: Percentage of stress required in fly

Memory category: static
Load category: not saved
Data type: integer
Attributes: limited access, field node, WITH MOVE ALONG
Limits: 1..100
Default: 50
S/W Version: 1.0
Description: It has the same purpose as \$STRESS_PER but is used with PATH segments. \$SEG_STRESS_PER doesn't apply to last node of path, \$STRESS_PER is used instead

12.396 \$SEG_TERM_TYPE: PATH segment termination type

Memory category: static
Load category: not saved
Data type: integer
Attributes: limited access, field node, WITH MOVE ALONG
Limits: FINE..JNT_COARSE
Default: NOSETTLE
S/W Version: 1.0
Description: It represents the termination type for a PATH segment. The variable has the same meaning as \$TERM_TYPE. This variable does not apply to the last node, in which case \$TERM_TYPE is used. If the standard field is not specified in the node declaration or in a WITH clause, the current program value is used

12.397 \$SEG_TOL: PATH segment tolerance

Memory category: static
Load category: not saved
Data type: real
Attributes: limited access, field node, WITH MOVE ALONG
Limits: none
Default:
S/W Version: 1.0
Description: It represents the tolerance for a PATH segment. \$SEG_TOL has the same meaning as \$TOL_FINE or \$TOL_COARSE depending on whether the termination type is FINE or COARSE. \$SEG_TOL does not apply to the last node, in which case the current arm value is used. If the standard field is not specified in the node definition (NODEDEF) or in a WITH clause, the current arm value is used

12.398 \$SEG_TOOL_IDX: PATH segment tool index

Memory category: static
Load category: not saved
Data type: integer
Attributes: limited access, field node, WITH MOVE ALONG
Limits: 0..7
Default: 0
S/W Version: 1.0
Description: Each PATH contains a frame table (FRM_TBL) of 7 positional elements that can be used either as a tool or a reference frame in a node which contains the MAIN_POS standard field. \$SEG_TOOL_IDX represents the FRM_TBL index to be used as the tool frame of a particular node; if \$SEG_TOOL_IDX is not included in the node definition or has value of zero (default value), the value of \$TOOL is used. As there is just one FRM_TBL in each path, for clearness it is suggested to use elements 1 to 3 inclusive for reference frames and elements 4 to 7 for tools. See example of [\\$SEG_REF_IDX: PATH segment reference index](#).

12.399 \$SEG_WAIT: PATH segment WAIT

Memory category: static
Load category: not saved
Data type: boolean
Attributes: limited access, field node, WITH MOVE ALONG
Limits: none
Default:
S/W Version: 1.0

Description: If this standard field is included in the node definition (NODEDEF) and the value is set TRUE for a particular node, the interpretation of the PATH motion is suspended; this means that motion for that node will complete but processing of following nodes will be suspended until the PATH is signalled. The default value for \$SEG_WAIT is FALSE. The SEGMENT WAIT condition handler event can be used to detect if a PATH is waiting; it is therefore suggested to globally enable the condition containing this event in order to monitor when the path is waiting on a segment.

The SIGNAL SEGMENT statement or action can be used for unsuspending the PATH
 PROGRAM segwait

```

        VAR pnt0001j, pnt0002j, pnt0003j : JOINTPOS FOR ARM[1]
        TYPE path_type = NODEDEF
            $MAIN_JNTP
            $SEG_WAIT
    ENDNODEDEF
    -- The three nodes of this path should either be taught or
    NODE_APPended
        VAR pth : PATH OF path_type
        i : INTEGER
        BEGIN
            -- This condition will signal the path nodes
    interpretation
        CONDITION [1] NODISABLE :
            WHEN SEGMENT WAIT pth DO
                .....
                SIGNAL SEGMENT pth
    ENDCONDITION
    ENABLE CONDITION [1]
    ...
    pth.NODE[1].$MAIN_JNTP := pnt0001j
    pth.NODE[1].$SEG_WAIT := FALSE
    ...
    pth.NODE[2].$MAIN_JNTP := pnt0002j
    -- the path interpretation will stop on this node
    pth.NODE[2].$SEG_WAIT := TRUE ...
    pth.NODE[3].$MAIN_JNTP := pnt0003j
    pth.NODE[3].$SEG_WAIT := FALSE
    ...
    MOVE ALONG pth
    ...
END segwait
    
```

12.400 \$SENSOR_CNVRSN: Sensor Conversion Factors

Memory category field of arm_data

Load category: arm. *Minor category* - sensor

Data type: array of real of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is a 6-elements array of REAL used to convert the values, read from the sensor, from the specific units (i.e. Amperes, Newtons, etc.) to millimeters. The first three elements are for corrections in X, Y, Z directions and the others are for rotations about X, Y, Z axes

12.401 \$SENSOR_ENBL: Sensor Enable

Memory category field of arm_data

Load category: arm. *Minor category* - sensor

Data type: boolean

Attributes: WITH MOVE

Limits none

Default:

S/W Version: 1.0

Description: It is used for enabling and disabling the Sensor Tracking features. It can either be used in a modal way or linked to a **MOVE Statement** after the WITH clause. It only has effect during programmed movements unless the SENSOR_TRK(ON) built-in has been executed

12.402 \$SENSOR_GAIN: Sensor Gains

Memory category field of arm_data

Load category: arm. *Minor category* - sensor

Data type: array of integer of one dimension

Attributes: none

Limits 0.100

Default: 00

S/W Version: 1.0

Description: It is a 6-elements array of INTEGER percentage values that define the speed to be used for the corrections. The first three elements are for corrections in X, Y, Z directions and the others are for rotations about X, Y, Z axes

12.403 \$SENSOR_OFST_LIM: Sensor Offset Limits

Memory category field of arm_data

Load category: arm. *Minor category* - sensor

Data type: array of real of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: It is a 2-elements array of REAL that defines the maximum distance allowed between the programmed trajectory and the robot under sensor control. The first element is for the translation (X, Y, Z) and is expressed in millimeters; the second is for rotations and is in degrees. When this limit is exceeded an error messages is issued and the robot is held

12.404 \$SENSOR_TIME: Sensor Time

Memory category field of arm_data
Load category: arm. *Minor category* - sensor
Data type: integer
Attributes: WITH MOVE
Limits 0..10000
Default: 0
S/W Version: 1.0
Description: It is an optional value that allows the user to define the period (in milliseconds) in which the corrections must be applied. The value 0 ms means that it is disabled and in this case the criterion of the distribution of the corrections is based on the speed programmed with \$SENSOR_GAIN. On the contrary, if \$SENSOR_TIME is specified then the corrections will be distributed during this period. Note that the speed programmed with \$SENSOR_GAIN will never be exceeded

12.405 \$SENSOR_TYPE: Sensor Type

Memory category field of arm_data
Load category: arm. *Minor category* - sensor
Data type: integer
Attributes: WITH MOVE
Limits 0..30
Default: 0
S/W Version: 1.0
Description: It is a coded value used for configuring Sensor Tracking. It defines three characteristics: where the information of the sensor can be read and what is the format of the data; in what frame of reference the corrections must be applied; if the corrections must be applied in a relative or absolute way.
 A list of valid values follows:

- 0: Sensor suspended;
- 1..4: Reserved
- 5: External sensor in TOOL frame, relative mode;
- 6: External sensor in UFRAME frame, relative mode;
- 7: External sensor in WORLD frame, relative mode;
- 8: External sensor in WEAVE frame, relative mode;
- 9: External sensor in TOOL frame, absolute mode;
- 10: External sensor in UFRAME frame, absolute mode;
- 11: External sensor in WORLD frame, absolute mode;
- 12: External sensor in WEAVE frame, absolute mode;
- 13..30: Reserved

External sensors are not integrated in the controller and can be managed by a PDL2 user program.

12.406 \$SERIAL_NUM: Serial Number of the board

Memory category: field of board_data
Load category: not saved
Data type: string
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: This is the serial number of the board

12.407 \$SFRAME: Sensor frame of an arm

Memory category: field of arm_data
Load category: arm. *Minor category - sensor*
Data type: position
Attributes: WITH MOVE
Limits none
Default:
S/W Version: 1.0
Description: It is a frame of reference like \$BASE, \$TOOL and \$UFRAME. It can be used to modify the frame in which the sensor corrections must be applied. It is defined referred to the tool frame or the user frame depending on \$SENSOR_TYPE

12.408 \$SING_CARE: Singularity care

Memory category: field of arm_data
Load category: arm. *Minor category - configuration*
Data type: boolean
Attributes: field node, WITH MOVE; MOVE ALONG
Limits TRUE..FALSE
Default: FALSE
S/W Version: 1.0
Description: It represents a flag that, if set, causes the motion to be held when the arm is near a singularity point. If not set, the speed will be reduced. The default value is FALSE

12.409 \$SM4_SAT_ADD_SCALE: Additional threshold for the SmartMove saturation

Memory category: field of arm_data
Load category: arm *Minor category* - configuration
Data type: array of INTEGER of one dimension
Attributes: property
Limits: 0..100
Default: 0
S/W Version: 1.0
Description: It represents the additional limit of saturation of the current foreseen by the dynamic model in correspondence of the the most significant fields. This limit is used for determining the times of acceleration and deceleration with the SmartMove algorithm in a critical path or an inverting fly.

12.410 \$SM4C_SAT_VEL_SCALE: thresholds for the joint speed saturation in Cartesian SmartMove

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of INTEGER of one dimension
Attributes: none
Limits: 15
Default: 0
S/W Version: 1.0
Description: It represents the limit of joint speed saturation in correspondence to the the most significant field used when planning a cartesian move; this limit is used for determining the time of acceleration and deceleration with the Cartesian SmartMove4 algorithm.

12.411 \$SM4C_STRESS_PER: Maximum Stress allowed in Cartesian SmartMove

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: INTEGER
Attributes: none
Limits: 0..100
Default: 0
S/W Version: 1.0

Description: Cartesian SmartMove4 increases, when possible, the values of some or all move parameters (speed, acceleration and jerk) of this percentage at the maximum.

12.412 \$SPD_OPT: Type of speed control

Memory category program stack

Load category: not saved

Data type: integer

Attributes: WITH MOVE; MOVE ALONG

Limits SPD_JNT..SPD_SM4C

Default: SPD_CONST

S/W Version: 1.0

Description: It represents the component of motion that governs the speed of a Cartesian motion. The following predefined constants can be used as values: SPD_JNT, SPD_CONST, SPD_LIN, SPD_ROT, SPD_SPN, SPD_AZI, SPD_ELV, SPD_ROLL, SPD_FIRST, SPD_SECOND, SPD_THIRD, SPD_AUX1, SPD_AUX2, SPD_AUX3, SPD_AUX4, SPD_PGOV and SPD_SM4C.

12.413 \$SREG: String registers - saved

Memory category static

Load category: controller. *Minor category - vars*

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: String registers that can be used by users instead of having to define variables. Also for Integer, Real, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.414 \$SREG_NS: String registers - not saved

Memory category static

Load category: not saved

Data type: array of string of one dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Non-saved String registers that can be used by users instead of having to define variables. Also for Integer, Real, Boolean, Jointpos, Position and Xtndpos datatypes there are saved and non-saved registers.

12.415 \$STARTUP: Startup program

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default: 'UD:\SETUP1'
S/W Version: 1.0
Description: It represents the program to be loaded and activated upon the controller startup. This variable can either be set via a PDL2 assignment statement or via the CONFIGURE CONTROLLER STARTUP (CCS) command. For clearing the value of this predefined variable, assign to it an empty string or do not specify the program name parameter to the CCS command. A CONFIGURE SAVE should then be executed

12.416 \$STARTUP_USER: Configuration file name

Memory category static
Load category: controller *Minor category - environment*
Data type: string
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: This username is used upon controller startup in case an implicit login has been defined (via the CONFIGURE CONTROLLER LOGIN STARTUP command). This saves the user from having to login from the user interface any time the system starts up

12.417 \$STRESS_PER: Stress percentage in cartesian fly

Memory category program stack
Load category: not saved
Data type: integer
Attributes: WITH MOVE; MOVE ALONG
Limits 1..100
Default: 50
S/W Version: 1.0
Description: It represents the maximum stress of the arm during a cartesian fly. It is only used if [\\$FLY_TYPE: Type of fly motion](#) is set to FLY_CART. It changes the fly trajectory and, if needed, reduces the programmed speed. The default value is 50%; 100% corresponds to passing through the programmed point at full speed.

12.418 \$STRK_END_N: User negative stroke end

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the maximum negative stroke for each axis of the arm. This value can be changed by the user, but has effect only if it is saved in the configuration file and the controller is restarted

12.419 \$STRK_END_P: User positive stroke end

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the maximum positive stroke for each axis of the arm. This value can be changed by the user, but has effect only if it is saved in the configuration file and the controller is restarted

12.420 \$STRK_END_SYS_N: System stroke ends

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the system negative stroke ends of the arm. This value cannot be changed by the user

12.421 \$STRK_END_SYS_P: System stroke ends

Memory category field of arm_data

Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: privileged read-write
Limits none
Default:
S/W Version: 1.0
Description: It represents the system positive stroke ends of the arm. This value cannot be changed by the user

12.422 \$SYNC_ARM: Synchronized arm of program

Memory category program stack
Load category: not saved
Data type: integer
Attributes: none
Limits 1..4
Default: uninitialized Integer
S/W Version: 1.0
Description: \$SYNC_ARM specifies the default synchronized arm for the SYNCMOVE clause. The user must initialize it before executing a SYNCMOVE clause without the arm specification otherwise an error is returned; in fact the default value for this variable is uninitialized.

12.423 \$SYS_CALL_OUT: Output lun for SYS_CALL

Memory category program stack
Load category: not saved
Data type: integer
Attributes: none
Limits none
Default: LUN_CRT
S/W Version: 1.0
Description: It represents the LUN (logical unit number) where the output of a SYS_CALL request is directed. This variable assumes by default the current value of \$DFT_LUN

12.424 \$SYS_CALL_STS: Status of last SYS_CALL

Memory category program stack
Load category: not saved
Data type: integer
Attributes: none
Limits 0..0
Default:

S/W Version: 1.0

Description: It represents the last error that occurred executing a SYS_CALL by the program.

12.425 \$SYS_CALL_TOUT: Timeout for SYS_CALL

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits 0..MAXINT

Default:

S/W Version: 1.0

Description: It represents the timeout for the duration of the SYS_CALL in msecs. A value of 0 means no timeout. Otherwise the SYS_CALL is completed in any case after a \$SYS_CALL_TOUT time.

12.426 \$SYS_ERROR: Last system error

Memory category static

Load category: not saved

Data type: integer

Attributes: none

Limits 0..0

Default: 0

S/W Version: 1.0

Description: It represents the last error that occurred in the system

12.427 \$SYS_ID: Robot System identifier

Memory category static

Load category: retentive, controller *Minor category - unique*

Data type: string

Attributes: privileged read-write

Limits none

Default:

S/W Version: 1.0

Description: It contains the Controller identifier, unique for each Controller, which should not be changed.

This identifier is used for locating Controller specific files eg. <\$SYS_ID>.C5G for configuration file.

12.428 \$SYS_OPTIONS: System options

Memory category: static
Load category: not saved
Data type: integer
Attributes: PDL2 read-only
Limits: none
Default:
S/W Version: 1.0
Description: This variable contains the definitions of the system features currently enabled on the Controller.

12.429 \$SYS_PARAMS: Robot system identifier

Memory category: static
Load category: retentive. *Minor category* - unique, parameter
Data type: array of integer of one dimension
Attributes: privileged read-write
Limits: none
Default:
S/W Version: 1.0
Description: This predefined variable contains system settings which is recommended not to be changed! The format and meaning of the data within these fields is reserved.

12.430 \$SYS_RESTART: System Restart Program

Memory category: static
Load category: controller. *Minor category* - environment
Data type: string
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It is the System restart program that the system tries to activate first. If the variable is empty or the program is not found, the \$RESTART program is being activated (if any).

12.431 \$SYS_STATE: State of the system

Memory category: static
Load category: not saved

<i>Data type:</i>	integer
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It represents a complete description of the state of the system and consists of the following mask:</p> <ul style="list-style-type: none"> – Bit 1: Indicates that all arms are in the Standby (energy saving) state – Bit 2: Indicates the presence of a jog movement in progress – Bit 3: Holdable progs(s) running – Bit 4: EMERGENCY(E-STOP) or SAFETY GATES(AUTO-STOP) or GENERAL STOP (GEN-STOP) alarm active – Bit 5: program WINC5G connected on a PC – Bit 6: Model of Teach Pendant (see also bit 23) – Bit 7: reserved – Bit 8: State selector on T1P – Bit 9: This bit remains set to 1 for the whole duration of the Power Failure recovery session. – Bit 10: reserved – Bit 11: TP Enabling device status: 1=pressed, 0=released. – Bit 12..13: Reserved – Bit 14: State selector on REMOTE – Bit 15: State selector on AUTO – Bit 16: State selector on T1 – Bit 17: HOLD signal from REMOTE source – Bit 18: HOLD button latched on TP – Bit 19: Reserved – Bit 20: DRIVE OFF signal from REMOTE source – Bit 21: DRIVE OFF button latched on TP – Bit 22: Reserved – Bit 23: TP connected – Bit 24: This bit is set to 1 if the system is in a ALARM state due to a fatal error (severity 13-15) – Bit 25: Indicates that the state of the system is REMOTE (see also bits 29). – Bit 26: START button pressed: <ul style="list-style-type: none"> • in AUTO: is set to 1 upon the first START button pressure when the system is in DRIVE ON. Returns to 0 when a HOLD or DRIVE OFF command is issued. • in PROG: this bit maintains the value of 1 as long as the START button remains pressed for the FORWARD function. – Bit 27: Indicates that the state of the system is ALARM. – Bit 28: Indicates that the state of the system is PROGR. – Bit 29: Indicates that the state of the system is AUTO (Auto-Teach or Local or Remote) (see also bit 25). – Bit 30: Indicates that the state of the system is HOLD. – Bit 31: DRIVEs status: 1 for DRIVE OFF, 0 for DRIVE ONBit 32: Reserved

12.432 \$TERM_TYPE: Type of motion termination

Memory category program stack

Load category: not saved

<i>Data type:</i>	integer
<i>Attributes:</i>	WITH MOVE; MOVE ALONG
<i>Limits</i>	FINE..JNT_COARSE
<i>Default:</i>	NOSETTLE
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It defines when the motion is to be terminated based on how close the arm is to its destination. Valid values are represented by the predefined constants NOSETTLE, COARSE, FINE, JNT_COARSE, and JNT_FINE. The default value is NOSETTLE.</p> <ul style="list-style-type: none"> – NOSETTLE: The move is considered complete when the trajectory generator has produced the last intermediate position for the arm. The actual position of the arm is not considered. – COARSE: The move is considered complete when the tool center point (TCP) has entered the tolerance sphere defined by the value of the predefined variable \$TOL_COARSE: Tolerance coarse. – FINE: The move is complete when the tool center point (TCP) has entered the tolerance sphere defined by the value of the predefined variable \$TOL_FINE. – JNT_COARSE: The move is considered complete when every joint has entered the joint tolerance sphere defined by the value of the predefined variable \$TOL_JNT_COARSE: Tolerance for joints [axis_num]. – JNT_FINE: The move is considered complete when every joint has entered the joint tolerance sphere defined by the value of the predefined variable \$TOL_JNT_FINE: Tolerance for joints [axis_num].

12.433 \$THRD_CEXP: Thread Condition Expression

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	0..0
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>This variable indicates which expression in a condition handler or WAIT FOR triggered. When an interrupt service routine is started as the action of a condition this variable is initialized to the condition expression number causing the condition to trigger. In the case of a WAIT FOR it indicates which expression in the WAIT FOR triggered. The number returned by the system matches the order in which the condition expressions are declared. There is one \$THRD_CEXP for each thread of execution. The user is only allowed to set this variable to 0. Please note that, in case of CONDITION, this variable is only set when entering the interrupt service routine used as action; it remains set to 0 otherwise.</p> <p>This variable is zeroed upon EXIT CYCLE.</p> <p>For example:</p> <pre> WAIT FOR HOLD OR EVENT 94 OR (\$FDIN[5]=TRUE) AND (\$FDOUT[5]=FALSE) IF \$THRD_CEXP = 3 THEN WRITE ('FDIN[5] IS TRUE AND FDOUT[5] IS FALSE', NL) ELSE IF \$THRD_CEXP = 2 THEN WRITE ('EVENT 94 event triggered', NL) </pre>

```

    ELSE -- $THRD_CEXP is 1
        WRITE ('The HOLD event triggered', NL)
    ENDIF
ENDIF

```

Note that, if a certain condition expression is deleted/added in a condition or wait for, the numbering could change if in such condition or wait for multiple expressions are present and depending also on the position of the being deleted/added one. In the above example, if the second condition expression (EVENT 94) is removed from the [WAIT FOR Statement](#), \$THRD_CEXP will assume the value of 1 if the HOLD event triggers and of 2 if the expression (\$FDIN[5] = TRUE) AND (\$FDOUT[5] = FALSE) triggers.

12.434 \$THRD_ERROR: Error of each thread of execution

Memory category: program stack

Load category: not saved

Data type: integer

Attributes: none

Limits 0..0

Default:

S/W Version: 1.0

Description: It represents the number of the last error in this thread of execution. There is a \$THRD_ERROR for every thread of execution. A new thread is started when an Interrupt Service Routine (routine that is an action of a condition) is activated. If the Interrupt Service Routine is an action of an error event condition, this variable is initialized to the error number causing the condition to trigger. Otherwise, it is initialized to zero.
This variable is zeroed upon EXIT CYCLE.

12.435 \$THRD_PARAM: Thread Parameter

Memory category: program stack

Load category: not saved

Data type: integer

Attributes: read-only

Limits none

Default:

S/W Version: 1.0

Description: It represents an associated parameter of the event which caused an Interrupt Service Routine (routine that is an action of a condition) to be executed. There is a \$THRD_PARAM for every thread of execution. A thread is started when an Interrupt Service Routine is activated. This variable is initialized to specific condition information. For motion events such as AT START, TIME BEFORE and for the event SEGMENT WAIT it is the number of the PATH node; for error events it is the number of the error; for EVENT events it is the number of the event. This variable is useful when one Interrupt Service Routine only is used for many different types of events.
This variable is zeroed upon EXIT CYCLE.

12.436 \$TIMER: Clock timer (in ms)

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents 1 msec timers available for use in PDL2 programs. The number of elements is dependant on the predefined variable [\\$NUM_TIMERS: Number of timers](#).
 Timer elements can be attached by different programs.
 See [ATTACH Statement](#) and [DETACH Statement](#).
 See also the other timers: [\\$PROG_TIMER_O: Program timer - owning context specific \(in ms\)](#), [\\$PROG_TIMER_OS: Program timer - owning context specific \(in seconds\)](#), [\\$PROG_TIMER_X: Program timer - execution context specific \(in ms\)](#), [\\$PROG_TIMER_XS: Program timer - execution context specific \(in seconds\)](#).

12.437 \$TIMER_S: Clock timer (in seconds)

Memory category: port
Load category: not saved
Data type: array of integer of one dimension
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: It represents 1 second timers for use in PDL2 programs. The number of elements is dependant on the predefined variable [\\$NUM_TIMERS: Number of timers](#).
 Timer elements can be attached by different programs.
 See [ATTACH Statement](#) and [DETACH Statement](#).
 See also the other timers: [\\$TIMER: Clock timer \(in ms\)](#), [\\$PROG_TIMER_O: Program timer - owning context specific \(in ms\)](#), [\\$PROG_TIMER_OS: Program timer - owning context specific \(in seconds\)](#), [\\$PROG_TIMER_X: Program timer - execution context specific \(in ms\)](#), [\\$PROG_TIMER_XS: Program timer - execution context specific \(in seconds\)](#).

12.438 \$TOL_ABST: Tolerance anti-bounce time

Memory category: field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: property
Limits: 0..5000

Default: 0
S/W Version: 1.0
Description: It represents the time, in milliseconds, in which the motion is defined to be completed after the tool center point (TCP) has first entered the tolerance sphere. Due to the effects of inertia, the TCP may enter the sphere of tolerance and immediately leave the sphere. The time is measured from the first time the tolerance sphere is entered.

12.439 \$TOL_COARSE: Tolerance coarse

Memory category field of arm_data
Load category: arm *Minor category* - configuration
Data type: real
Attributes: property, WITH MOVE; MOVE ALONG
Limits 0.5 .. 5.0
Default: 0.5
S/W Version: 1.0
Description: It represents the spherical region, in millimeters, for defining the termination of motion when [\\$TERM_TYPE: Type of motion termination](#) is set to COARSE.

12.440 \$TOL_FINE: Tolerance fine

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: real
Attributes: property, WITH MOVE; MOVE ALONG
Limits 0.1 .. 5.0
Default: 0.1
S/W Version: 1.0
Description: It represents the spherical region, in millimeters, for defining the termination of motion when [\\$TERM_TYPE: Type of motion termination](#) is set to FINE.

12.441 \$TOL_JNT_COARSE: Tolerance for joints

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: array of real of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the tolerance in degrees for each rotating joint and in millimeters for each

traslating joint. This parameter is used when [\\$TERM_TYPE: Type of motion termination](#) is JNT_COARSE. There are no limit checks performed on these values. Any change to the value of this variable immediately takes effect on the next movement.

12.442 \$TOL_JNT_FINE: Tolerance for joints

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - configuration</i>
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the tolerance in degrees for each rotating joint and in millimeters for each translating joint. This parameter is used when \$TERM_TYPE: Type of motion termination is JNT_FINE. There are no limit checks performed on these values. Any change to the value of this variable immediately takes effect on the next movement.

12.443 \$TOL_TOUT: Tolerance timeout

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - configuration</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	property
<i>Limits</i>	0..20000
<i>Default:</i>	2000
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the timeout for settling within the specified sphere of tolerance, in milliseconds. If the arm does not come within the tolerance sphere within the timeout, a warning message is displayed.

12.444 \$TOOL: Tool of arm

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category - chain</i>
<i>Data type:</i>	position
<i>Attributes:</i>	WITH MOVE; MOVE ALONG
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the tool dimension and orientation

12.445 \$TOOL_CNTR: Tool center of mass of the tool

Memory category: field of arm_data
Load category: arm. *Minor category* - chain
Data type: position
Attributes: WITH MOVE; MOVE ALONG
Limits: none
Default:
S/W Version: 1.0
Description: It represents the position of the barycenter of the tool. This is the point where could be concentrated the total mass of the tool (\$TOOL_MASS). Only the first three components of the POSITION (X, Y, Z) are used (the Euler angles are not significant)

12.446 \$TOOL_INERTIA: Tool Inertia

Memory category: field of arm_data
Load category: arm. *Minor category* - chain
Data type: array of real of one dimension
Attributes: WITH MOVE
Limits: none
Default:
S/W Version: 1.0
Description: This array represents the tensor of inertia of the tool determined by the algorithm of payload identification

12.447 \$TOOL_MASS: Mass of the tool

Memory category: field of arm_data
Load category: arm. *Minor category* - chain
Data type: real
Attributes: WITH MOVE; MOVE ALONG
Limits: none
Default:
S/W Version: 1.0
Description: It represents the tool mass expressed in Kg

12.448 \$TOOL_RMT: Fixed Tool

Memory category: field of arm_data

Load category: arm. *Minor category - chain*
Data type: boolean
Attributes: WITH MOVE
Limits none
Default:
S/W Version: 1.0
Description: If set to TRUE, the remote tool system is enabled. In this case \$TOOL represents the position of the remoted tool center point with respect to the world frame of reference; \$UFRAME is defined with respect to the flange frame of reference

12.449 \$TOOL_XTREME: Extreme Tool of the Arm

Memory category field of arm_data
Load category: arm. *Minor category - chain*
Data type: position
Attributes: field node, WITH MOVE, MOVE ALONG
Limits none
Default:
S/W Version: 1.0
Description: it represents the more far away point reached by the tool: if not implicitly declared, it is the same by default of \$TOOL

12.450 \$TP_ARM: Teach Pendant current arm

Memory category static
Load category: not saved
Data type: integer
Attributes: read-only
Limits 1..4
Default: 1
S/W Version: 1.0
Description: It represents the arm currently selected on the teach pendant and displayed on the status line of the system screen

12.451 \$TP_GEN_INCR: Incremental value for general override

Memory category static
Load category: not saved
Data type: integer
Attributes: none

Limits 0..10
Default: 5
S/W Version: 1.0
Description: It represents the general override increment currently being used at the teach pendant

12.452 \$TP_MJOG: Type of TP jog motion

Memory category static
Load category: not saved
Data type: integer
Attributes: none
Limits 0..3
Default: 0
S/W Version: 1.0
Description: It represents the type of motion currently enabled on the teach pendant. Possible values are:

- 0: Joint
- 1: Base
- 2: Tool
- 3: User frame

12.453 \$TP_ORNT: Orientation for jog motion

Memory category field of arm_data
Load category: arm. *Minor category* - configuration
Data type: integer
Attributes: none
Limits WRIST_JNT..RPY_WORLD
Default: RPY_WORLD
S/W Version: 1.0
Description: It represents the type of orientation used when jog keys 4, 5, and 6 are pressed and the type of motion selected on the teach pendant is not JOINT. If \$TP_ORNT is set to RPY_WORLD, these keys determine the rotation along axes X, Y, Z respectively in the selected frame of reference (BAS, TOL or USR). If \$TP_ORNT is set to WRIST_JNT, these keys determine the movement of axes 4, 5 and 6 in joint mode, regardless of the active frame of reference. Note that \$TP_ORNT can be changed either from PDL2 or from the teach pendant

12.454 \$TP_SYNC_ARM: Teach Pendant's synchronized arms

Memory category static

Load category: not saved
Data type: array of integer of one dimension
Attributes: read-only
Limits 0..4
Default:
S/W Version: 1.0
Description: It is an array of 2 integers where the first element contains the main arm and the second element contains the synchronized arm. These are selected via the Teach Pendant; in the Status Bar of the Teach Pendant a marker ‘<’ indicates which Arm is currently set for jogging (it is the one on the left of the marker).
Examples:
Arm: 2<1 means that Arm 2 is the main Arm and it is selected for jogging
Arm: 1<2 means that Arm 1 is the main Arm and it is selected for jogging.

For further information about how to select an Arm, see **C5G Control Unit Use Manual, Right Menu.**

12.455 \$TRK_TBL: Tracking application table data



NOTE

To better understand the below listed variables, please read chap.[Conveyor Tracking \(optional feature\)](#) in [Motion Programming manual](#).

Memory category dynamic
Load category: controller *Minor category - tracking*
Data type: array of \$trk_tbl of one dimension
Attributes: none
Limits none
Default: none
S/W Version: 1.0
Description: \$TRK_TBL is an array of 10 tracking schedules with each schedule containing the \$TT_xxx fields:
 – \$TT_APPL_ID: Tracking application identifier
 – \$TT_ARM_MASK: Tracking arm mask
 – \$TT_FRAMES: Array of POSITIONs for tracking application
 – \$TT_I_PRMS: Integer parameters for the tracking application
 – \$TT_PORT_IDX: Port Index for the tracking application
 – \$TT_PORT_TYPE: Port Type for the tracking application
 – \$TT_R_PRMS: Real parameters for the tracking application.

12.456 \$TT_APPL_ID: Tracking application identifier

Memory category field of trk_tbl
Load category: controller *Minor category - tracking*
Data type: integer

Attributes: property
Limits none
Default: none
S/W Version: 1.0
Description: Is the application ID for tracking feature.

12.457 \$TT_ARM_MASK: Tracking arm mask

Memory category field of trk_tbl
Load category: controller *Minor category* - tracking
Data type: integer
Attributes: property
Limits none
Default: 0
S/W Version: 1.0
Description: Is the arm mask which is in use or enabled for the tracking application.

12.458 \$TT_FRAMES: Array of POSITIONs for tracking application

Memory category field of trk_tbl
Load category: controller *Minor category* - tracking
Data type: array[3] of position
Attributes: property
Limits none
Default: none
S/W Version: 1.0
Description: Is an array of the allowed frames for the tracking application. The meaning is different for different applications.

12.459 \$TT_I_PRMS: Integer parameters for the tracking application

Memory category field of trk_tbl
Load category: controller *Minor category* - tracking
Data type: array[20] of integer
Attributes: property
Limits none
Default:
S/W Version: 1.0

Description: Is an array of integer parameters used for the tracking application. The meaning is different for different applications.

12.460 \$TT_PORT_IDX: Port Index for the tracking application

Memory category field of trk_tbl

Load category: controller *Minor category* - tracking

Data type: array[5] of integer

Attributes: property

Limits 0..2048

Default: 0

S/W Version: 1.0

Description: Is an array of port index strictly connected with [\\$TT_PORT_TYPE: Port Type for the tracking application](#).

12.461 \$TT_PORT_TYPE: Port Type for the tracking application

Memory category field of trk_tbl

Load category: controller *Minor category* - tracking

Data type: array[5] of integer

Attributes: property

Limits none

Default: 0

S/W Version: 1.0

Description: Is an array of ports allowed for the configured tracking application. The meaning is different for different applications. See also [\\$TT_PORT_IDX: Port Index for the tracking application](#).

12.462 \$TT_R_PRMS: Real parameters for the tracking application

Memory category field of trk_tbl

Load category: controller *Minor category* - tracking

Data type: array[20] of real

Attributes: property

Limits none

Default: none

S/W Version: 1.0

Description: Is an array of real parameters used for the tracking application. The meaning is different for different applications.

12.463 \$TUNE: Internal tuning parameters

Memory category static

Load category: controller. *Minor category* - shared

Data type: array of integer of one dimension

Attributes: none

Limits: none

Default:

S/W Version: 1.0

Description: It represents internal tuning values. Some of the more useful elements are:

[1]: Condition scan time. The default value is 20 msecs. The minimum limit is 10 msecs

[2..5]: reserved

[6]: Drives answer timeout in msecs. The default value is 4000 (4 seconds)

[7]: Tasks answer timeout in msecs. The default value is 3000 (3 seconds)

[8]: reserved

[9]: Drives enable delay in msecs. The default value is 400

[10]: Slow speed, brake on/off delay in msecs. The default value is 2000 (2 seconds)

[11]: teach pendant connection delay in msecs. The default value is 3000 (3 seconds)

[12]: reserved

[13]: Alarm exclusion delay in msecs. The default value is 60000 (60 seconds)

[14]: Startup delay time for REMOTE command in msecs. The default value is 5000 (5 seconds)

[15]: DISPLAY command refresh time in msecs. The default value is 1000 (1 second)

[16]: Maximum number of lines available to the DISPLAY group of commands when issued on the Terminal window of WinC5G program that runs on the PC. The default value is 16 lines

[17]: Maximum number of lines available to the DISPLAY group of commands when issued on the programming terminal. The default value is 10 lines

[18]: reserved

[19]: Timeout for TP backlight. The default value is 180 seconds (3 minutes).

Allowed values:

-1 - the TP will never switch off (NOT RECOMMENDED because this could cause the display life to be reduced);

0 - it is automatically set by the system to 180.

[20]: Refresh time, in msecs, for the FOLLOW command of the PROGRAM EDIT and MEMORY DEBUG environments. The default value is every half second

[21]: Status line refresh time in msecs. The default value is 500

[22]: Automatic logout timeout (in ms). The default value is 0 for indicating No timeout

[23]: reserved

[24]: During PowerFailureRecovery, this is the lag in some error logging. The default is 800 msecs.

[25]: reserved

- [26]: Software timer that activates the motors brake when time expires after the controlled emergency. The default value is 1400 msecs
- [27]: Energy saving timeout expressed in seconds. The default is 120
- [28]: reserved
- [29]: size, in bytes, of the stack to be allocated for PDL2 programs. The default is 1000
- [30]: Maximum size, in bytes, for read-ahead buffer. The default is 4096
- [31]: Maximum execution time, in msecs, for the calibration program. The default is 10000
- [32..41]: reserved
- [42]: Default stack size, in bytes, for EXECUTE command. The default is 300
- [43..50]: reserved
- [54]: Number of default characters read. Default 256

12.464 \$TURN_CARE: Turn care

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	boolean
<i>Attributes:</i>	property, WITH MOVE; MOVE ALONG
<i>Limits</i>	none
<i>Default:</i>	OFF
<i>S/W Version:</i>	1.0
<i>Description:</i>	The Cartesian trajectory usually follows the shortest path for the joints, so the configuration string of the final position reached can be different from the one present in the MOVE statement. \$TURN_CARE represents a flag to determine whether the system must check the configuration turn numbers. If the flag is FALSE no check is done. If the flag is TRUE the system sends an error message if the Cartesian trajectory tries to move the robot to a final position having a configuration number of turns different from the number of turns present in the MOVE statement

12.465 \$TX_RATE: Transmission rate

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm <i>Minor category</i> - configuration
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	It represents the relationship between the turn of the motor and the turn of the axes. For a translational axis this is represented in millimeters

12.466 \$UDB_FILE: Name of UDB file

Memory category static
Load category: controller. *Minor category - environment*
Data type: string
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: It represents the name of the .UDB file to be loaded.

12.467 \$UFRAME: User frame of an arm

Memory category field of arm_data
Load category: arm. *Minor category - chain*
Data type: position
Attributes: WITH MOVE
Limits none
Default:
S/W Version: 1.0
Description: It represents the location and orientation of the user frame

12.468 \$VERSION: Software version

Memory category static
Load category: not saved
Data type: real
Attributes: read-only
Limits none
Default:
S/W Version: 1.0
Description: It represents the current version of the system software

12.469 \$VP2_SCRN_ID: Executing program VP2 Screen Identifier

Memory category program stack
Load category: not saved
Data type: integer

Attributes: read-only
Limits 0..MAXINT
Default:
S/W Version: 1.0
Description: This variable contains the identifier of the VP2 Screen, created by the executing program, using **VP2_SCRN_CREATE** (see **VP2 - Visual PDL2** manual - chapter 6 - **VP2 Built-ins**).

12.470 \$VP2_TOUT: Timeout value for asynchronous VP2 requests

Memory category program stack
Load category: not saved
Data type: integer
Attributes: none
Limits 0..MAXINT
Default:
S/W Version: 1.0
Description: This variable contains the timeout value for asynchronous VP2 calls; a value of 0 means an indefinite wait.

12.471 \$VP2_TUNE: Visual PDL2 tuning parameters

Memory category static
Load category: controller. *Minor category* - shared
Data type: array of integer of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: This variable is used for setting Visual PDL2 internal parameters.

12.472 \$WEAVE_MODALITY: Weave modality

Memory category field of prog_arm_data
Load category: not saved
Data type: integer
Attributes: field node, WITH MOVE; MOVE ALONG
Limits none
Default: 0
S/W Version: 1.0

Description: This variable identifies the modality in which the shape of the wave is generated.

- Setting it to 0 (default value), the wave shape is created using the traversal speed;
- setting it to 1, the wave shape is created using the length of the weave.

It is not affected by changing other parameters (override, speed, etc.).

12.473 \$WEAVE_MODALITY_NOMOT: Weave modality (only for no arm motion)

Memory category: field of crnt_data

Load category: not saved

Data type: integer

Attributes: none

Limits: none

Default:

S/W Version: 1.0

Description: This variable identifies the modality in which the shape of the wave is generated. Setting it to 0 (default), the wave shape is created using the traversal speed; setting it to 1, the wave shape is created using the length of the weave. It is not affected by the changing of other parameters such as override, speed, etc. It should only be used if the weaving is performed without arm motion. It replaces \$WEAVE_MODALITY in absence of arm movement.

12.474 \$WEAVE_NUM: Weave table number

Memory category: field of prog_arm_data

Load category: not saved

Data type: integer

Attributes: field node, WITH MOVE; MOVE ALONG

Limits: 0..NUM_WEAVES

Default: 0

S/W Version: 1.0

Description: It represents the weave table (\$WEAVE_TBL) element to be used in the next motions. The value of zero disables the weaving

12.475 \$WEAVE_NUM_NOMOT: Weave table number (only for no arm motion)

Memory category: field of crnt_data

Load category: not saved

Data type: integer

Attributes: none

Limits: 0..10

Default: 0
S/W Version: 1.0
Description: This field of \$CRNT_DATA indicates the weave table (\$WEAVE_TBL) element to be considered in case weaving is performed without arm motion. It replaces \$WEAVE_NUM in absence of arm movement. The value of zero disables the weaving.

12.476 \$WEAVE_PHASE: Index of the Weaving Phase

Memory category field of crnt_data
Load category: not saved
Data type: integer
Attributes: PDL2 read-only
Limits none
Default:
S/W Version: 1.0
Description: it contains the index of the active weaving phase

12.477 \$WEAVE_TBL: Weave table data

Memory category dynamic
Load category: not saved
Data type: array of weave_tbl of one dimension
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: \$WEAVE_TBL is an array [\$NUM_WEAVES] of weaving schedules with each schedule containing the \$WV_xxx fields. To superimpose weave on a motion, the specific weave table element has to be specified in the \$WEAVE_NUM variable

12.478 \$WEAVE_TYPE: Weave type

Memory category field of prog_arm_data
Load category: not saved
Data type: integer
Attributes: field node, WITH MOVE; MOVE ALONG
Limits 0..NAX_ARM
Default: 0
S/W Version: 1.0

Description: This variable selects the kind of weaving between the cartesian mode and the joint mode. Set this variable to zero to enable the cartesian weaving. Otherwise, in case of joint weaving, values from 1 to 8 select the axis on which weaving is done.

12.479 \$WEAVE_TYPE_NOMOT: Weave type (only for no arm motion)

Memory category field of crnt_data

Load category: not saved

Data type: integer

Attributes: none

Limits 0..10

Default: 0

S/W Version: 1.0

Description: This field of \$CRNT_DATA selects the kind of weaving between the cartesian mode and the joint mode. Set this variable to zero to enable the cartesian weaving. Otherwise, in case of joint weaving, values from 1 to 8 select the axis on which weaving is performed. It should only be used if the weaving is performed without arm motion. It replaces \$WEAVE_TYPE in absence of arm movement.

12.480 \$WFR_IOTOUT: Timeout on a WAIT FOR when IO simulated

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits 0..2147483647

Default: 0

S/W Version: 1.0

Description: It represents the timeout in milliseconds for a WAIT FOR when the port is simulated/forced. The default value is zero, meaning that the WAIT FOR will not timeout

12.481 \$WFR_TOUT: Timeout on a WAIT FOR

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits 0..2147483647

Default: 0

S/W Version: 1.0

Description: It represents the timeout in milliseconds for a WAIT FOR. The default value is zero, meaning that the WAIT FOR will not timeout.

12.482 \$WORD: WORD data

Memory category port

Load category: not saved

Data type: array of integer of one dimension

Attributes: none

Limits: none

Default:

S/W Version: 1.0

Description: This structure is an analogue port array; the size of each element is 16 bits. For further information see also [par. 5.4.2 \\$WORD on page 119](#).

12.483 \$WRITE_TOUT: Timeout on a WRITE

Memory category program stack

Load category: not saved

Data type: integer

Attributes: none

Limits: 0..2147483647

Default: 0

S/W Version: 1.0

Description: It represents the timeout in milliseconds for an asynchronous WRITE. The default value is zero, meaning that the WRITE will not timeout. This is useful when a message has to be sent in a specific time period.

12.484 \$WV_AMPLITUDE: Weave amplitude

Memory category field of weave_tbl

Load category: not saved

Data type: real

Attributes: none

Limits: none

Default: 0

S/W Version: 1.0

Description: It represents the weaving amplitude (in mm) in the direction perpendicular to the trajectory in case of circular weaving (it is the longest semi-axis of the ellipse representing the shape of the weave).

12.485 \$WV_AMP_PER_RIGHT/LEFT: Weave amplitude percentage

Memory category: field of weave_tbl
Load category: not saved
Data type: integer
Attributes: WITH MOVE
Limits 0..1000
Default: 100%
S/W Version: 1.0
Description: The weaving amplification is a parameter for variable weaving width. It is a percentage parameter representing amplitude at the end of the current trajectory; the amplitude will change linearly from the initial to the final value. This variable is only used at the beginning of the current motion segment. It is not dynamically read during the motion.

12.486 \$WV_CNTR_DWL: Weave center dwell

Memory category: field of weave_tbl
Load category: not saved
Data type: integer
Attributes: none
Limits 0.10000
Default: 0
S/W Version: 1.0
Description: If \$WEAVE_MODALITY is set to 0, it represents the dwell time (in milliseconds) of the wave center.
 If \$WEAVE_MODALITY is set to 1, it represents the distance (in millimeters), on the trajectory, of the wave center.

12.487 \$WV_DIRECTION_ANGLE: Weave angle direction

Memory category: field of weave_tbl
Load category: not saved
Data type: real
Attributes:
Limits -180..+180
Default: 0
S/W Version: 1.0

Description: This represents the angle (included between -180 and + 180) allowed for the weaving direction in case of weaving with the robot held or in case of weaving plane solid with the tool.
The default direction (0 angle) corresponds to the X direction of the TOOL.

12.488 \$WV_END_DWL Weave end dwell

Memory category field of weave_tbl
Load category: not saved
Data type: Integer
Attributes: none
Limits 0 10000
Default: 0
S/W Version: 1.0
Description: If \$WEAVE_MODALITY is set to 1, it represents the the distance (in millimeters), on the trajectory, of the end-hand side of the wave. If \$WEAVE_MODALITY is set to 2, it represents the the dwell time (in milliseconds) of the end-hand side of the wave. If the \$WEAVE_MODALITY is set to 0, it has no effect.

12.489 \$WV_LEFT_AMP: Weave left amplitude

Memory category field of weave_tbl
Load category: not saved
Data type: real
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: This represents the amplitude of the left-hand side of the weave. It is expressed in millimeters in case of cartesian weaving; in case of joint weaving it is expressed in degrees or millimeters depending on whether the selected axis is rotational or translational, respectively

12.490 \$WV_LEFT_DWL: Weave left dwell

Memory category field of weave_tbl
Load category: not saved
Data type: integer
Attributes: none
Limits 0..10000
Default: 0
S/W Version: 1.0

Description: If \$WEAVE_MODALITY is set to 0, it represents the dwell time (in milliseconds) of the left-hand side of the wave.
 If \$WEAVE_MODALITY is set to 1, it represents the distance (in millimeters), on the trajectory, of the left-hand side of the wave.

12.491 \$WV_LENGTH_WAVE: Wave length

Memory category field of weave_tbl
Load category: not saved
Data type: integer
Attributes: none
Limits 0..10000
Default: 0
S/W Version: 1.0
Description: It represents the length, in millimeters, of the weaving shape wave

12.492 \$WV_ONE_CYCLE: Weave one cycle

Memory category field of weave_tbl
Load category: not saved
Data type: boolean
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: If set to TRUE the weave will only perform one cycle

12.493 \$WV_PLANE: Weave plane angle

Memory category field of weave_tbl
Load category: not saved
Data type: real
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: In the cartesian weaving, this variable is an angle used for assigning the weaving direction. It is calculated with respect to the approach vector (-180 , +180 degree) at the first point of the weave. A value of zero means the plane of the weave is perpendicular to the tool approach and the line of the trajectory. This variable is only used at the beginning of non-continuous Cartesian motions and is not dynamically read during the motion. In the joint weaving mode, this variable is ignored

12.494 \$WV_PLANE_MODALITY: Weave plane modality

Memory category: field of weave_tbl
Load category: not saved
Data type: integer
Attributes: none
Limits: 0..1
Default: 0
S/W Version: 1.0
Description: It represents the modality in which it is calculated the weaving plane.
If this variable is set to 0 (default), the weaving plane is maintained constant in respect to the sequence of trajectories linked in fly.
If this variable is set to 1, the weaving plane is calculated on the trajectory and is solidal with the tool and perpendicular to this.

12.495 \$WV_RADIUS: Weave radius

Memory category: field of weave_tbl
Load category: not saved
Data type: real
Attributes: none
Limits: none
Default: 0
S/W Version: 1.0
Description: It represents the amplitude (in mm) of the weaving in the trajectory direction in case of circular weaving (it is the shortest semiaxis of the ellipse representing the shape of the weave).

12.496 \$WV_RIGHT_AMP: Weave right amplitude

Memory category: field of weave_tbl
Load category: not saved
Data type: real
Attributes: none
Limits: none
Default:
S/W Version: 1.0
Description: This represents the amplitude of the right-hand side of the weave. It is expressed in millimeters in case of cartesian weaving; in case of joint weaving it is expressed in degrees or millimeters depending on whether the selected axis is rotational or translational, respectively

12.497 \$WV_RIGHT_DWL: Weave right dwell

Memory category: field of weave_tbl
Load category: not saved
Data type: integer
Attributes: none
Limits 0..10000
Default: 0
S/W Version: 1.0
Description: If \$WEAVE_MODALITY is set to 0, it represents the dwell time (in milliseconds) of the right-hand side of the wave.
 If \$WEAVE_MODALITY is set to 1, it represents the distance (in millimeters), on the trajectory, of the right-hand side of the wave.

12.498 \$WV_SMOOTH: Weave smooth enabled

Memory category: field of weave_tbl
Load category: not saved
Data type: boolean
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: A sinusoid-like wave shaped weave as opposed to a trapezoidal-like wave shaped weave can be obtained by using this smoothing flag. Setting \$WV_SMOOTH to TRUE slightly reduces the frequency. Changing this variable during the motion will effect the weave

12.499 \$WV_SPD_PROFILE: Weave speed profile enabled

Memory category: field of weave_tbl
Load category: not saved
Data type: boolean
Attributes: none
Limits none
Default:
S/W Version: 1.0
Description: the trasversal speed is reached on a deceleration/acceleration threshold which is determined accordingly to the characterization of : the interested axis (joint weaving) ; the linear velocity (Cartesian weaving). This variable has no effect if \$WV_SMOOTH flag is set to TRUE.

12.500 \$WV_TRV_SPD: Weave transverse speed

<i>Memory category</i>	field of weave_tbl
<i>Load category:</i>	not saved
<i>Data type:</i>	real
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	This variable represents the transverse speed of the tool across the weave. It is expressed in meters/second in case of cartesian weaving; in case of joint weaving is expressed in radians/second or meters/second depending on whether the selected axis is rotational or translational respectively

12.501 \$WV_TRV_SPD_PHASE: Weave transverse speed phase

<i>Memory category</i>	field of weave_tbl
<i>Load category:</i>	not saved
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	If \$WEAVE_MODALITY is set to 0, it hasn't effect. If \$WEAVE_MODALITY is set to 1, it represents the the distance (in millimeters) on the trajectory, of the left/right-hand side of the wave. If \$WEAVE_MODALITY is set to 2, it represents the speed on the trajectory, of the left/right-hand side of the wave. It is expressed in meters/second in case of cartesian weaving; in case of joint weaving is expressed in radians/second or meters/second depending on whether the selected axis is rotational or translational respectively

12.502 \$XREG: Xtndpos registers - saved

<i>Memory category</i>	statics
<i>Load category:</i>	controller. <i>Minor category - vars</i>
<i>Data type:</i>	array of xtndpos of two dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>Default:</i>	
<i>S/W Version:</i>	1.0
<i>Description:</i>	Xtndpos registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Jointpos and Position datatypes there are saved and non-saved registers.

12.503 \$XREG_NS: Xtndpos registers - not saved

Memory category: static

Load category: not saved

Data type: array of xtndpos of two dimension

Attributes: none

Limits none

Default:

S/W Version: 1.0

Description: Non-saved Xtndpos registers that can be used by users instead of having to define variables. Also for Integer, Real, String, Boolean, Jointpos and Position datatypes there are saved and non-saved registers.

13. POWER FAILURE RECOVERY

For information about the current topic, please call Comau.

14. TRANSITION FROM C4G TO C5G CONTROLLER

14.1 Introduction

This chapter reports a list of the main differences in PDL2 language (and not only in the language) that should be considered before running on C5G Control Unit a PDL2 program written in the past for C4G Control Unit.

Detailed information is supplied about the following topics:

- How to run C4G programs on C5G Controller
- Predefined Variables list
- DV_CNTRL
- Environment variables on the PC
- Built-in routines and functions.

14.2 How to run C4G programs on C5G Controller

Programs which have been developed on the C4G controller cannot be directly run on the C5G without any modification. They should, at least and in the best of cases, pass through a translation process.

However, the user can modify his programs following what stated in this chapter and then run them on the C5G controller.

Transform the files from ASCII to binary, which means translate them from .COD to .PDL and from .var to .LSV.

Then, on the C5G controller; translate them back (from .PDL to .COD, from .LSV, to .VAR).

In some cases and depending from the statements inside the programs, errors can occur.

For solving them, it is necessary to identify the errors and correct them by removing what is no more supported on the C5G. Possible causes of the errors: a predefined variable is accessed but that variable does not exist or has been renamed or if a field of a different parent. A build-in routine has changed the name or is no more supported. A device does not exist anymore.



The current chapter identifies the main differences in the PDL2 language, between C4G and C5G platform. Please, also refer to the chapters in which the related feature is described.

14.3 Predefined Variables list

Information are supplied about the following topics:

- Variables which have been renamed or moved to other structures
- Removed Predefined Variables

- New Predefined Variables

14.3.1 Variables which have been renamed or moved to other structures

- \$B_ALONG_1D_NS is no more a field of \$BOARD_DATA but is an independent predefined variable.
- \$DSA_DATA [num_dsas] is changed to \$LOOP_DATA [num_arms]. Related fields:
 - \$D_ALONG_1D[200] is changed to \$L_ALONG_1D[200];
 - \$D_ALONG_2D[1, 10] is changed to \$L_ALONG_2D[240, 10]
 - \$D_AREAL_1D[24] is changed to \$LD_AREAL_1D[24];
 - \$D_AREAL_2D[10, 10] is changed to \$LL_AREAL_2D[480, 10]

The meaning of the fields could have changed. Privileged user only is allowed to know them.

- \$IPERIOD is now an independent variable.
- \$C4G_IMPORT is changed to \$CRC_IMPORT
- \$C4G_PA is changed to \$CRC_PA
- \$C4G_RULES is changed to \$CRC_RULES
- \$C4G_STS is changed to \$SYS_ACC_STS
- \$C4G_TOUT is changed to \$SYS_ACC_TOUT
- \$MCP_DATA[].HDIN_SUSP is changed to \$CRNT_DATA[].HDIN_SUSP
- \$SDIN is changed to \$SDI
- \$SDOUT is changed to \$SDO
- \$SDIN32 is changed to \$SDI32
- \$SDOUT32 is changed to \$SDO32

14.3.2 Removed Predefined Variables

The following predefined variables, present on the C4G, are no longer supported on the C5G:

- \$BOARD_DATA, \$B_ALONG_1D, \$M_ALONG_1D, \$R_ALONG_1D_A, \$R_ALONG_1D_B
- \$CIO_CAN, \$CIO_SYS_CAN, \$CIO_AIN, \$CIO_AOUT, \$CIO_CROSS, \$CIO_DIN, \$CIO_DOUT, \$CIO_FMI, \$CIO_FMO, \$CIO_GIN, \$CIO_GOUT, \$CIO_IN_APP, \$CIO_OUT_APP, \$CIO_SDIN, \$CIO_SDOUT
- \$CONV_ZERO, \$CONV_WIN, \$CONV_TYPE, \$CONV_CNFG, \$CONV_BASE, \$CONV_SPD_LIM, \$CONV_ACC_LIM, \$CONV_TBL, \$CT_TX_RATE, \$CT_JNT_MASK, \$CT_SCC, \$CT_RES, \$CT_RADIUS, \$CT_SHEET_DEPTH, \$CT_CAVE_ANGLE, \$CT_SHOULDER_RADIUS, \$CT_CAVE_WIDTH, \$CT_KNIFE_RADIUS, \$CT_DELAY, \$CT_ARMA_FILTER, \$CT_MAX_BEND_ANGLE
- \$CUSTOM_ARM_ID
- \$D_AXES, \$D_CTRL, \$D_MTR
- \$CURR_FILTER, \$DYN_FILTER, \$FORCE_FILTER, \$SS_JNT_VEL_THRS, \$SS_KD, \$SS_KTRANS, \$SS_MASS, \$SS_MOV_THRS, \$SS_SO_FORCE_THRS, \$SS_SO_TORQUE_THRS, \$SS_TOMOV_TIME,

- \$SS_TOSTOP_TIME, \$SS_VEL_SO_THRS, \$THRS_INMOV, \$THRS_INSTOP,
\$TOOL_FRICTION, \$VEL_FILTER,
- \$FB_CNFG, \$FB_INIT, \$FBP_TBL, \$FB_ADDR, \$FB_MA_INIT, \$FB_MA_SLVS,
\$FB_MA_SLVS_NAME, \$FB_MA_SLVS_PRMS, \$FB_MA_SLVS_STR,
\$FB_SLOT, \$FB_SL_INIT, \$FB_TYPE
- \$D_HDIN_SUSP, \$HDOUT (never used)
- \$GIN, \$GOUT
- \$LOG_TO_DSA, \$LOG_TO_PHY
- \$MCP_DATAs, including \$REF_ARMS
- \$NUM_DSAS, \$NUM_MCP_DATAS, \$NUM_MCPS
- \$PPP_INT
- \$PROG_UADDR, \$PROG_UBIT, \$PROG_UBYTE, \$PROG_ULEN,
\$PROG ULONG, \$PROG_UWORD
- \$SWIM_ADDR, \$SWIM_CNFG, \$SWIM_INIT
- \$SYS_PROT, \$SYS_PROT_STATE
- \$UDB_GROUP
- \$USER_ADDR, \$USER_BIT, \$USER_BYTE, \$USER_LEN, \$USER_LONG,
\$USER_WORD.

14.3.3 New Predefined Variables

The following predefined variables have been created for C5G environment:

- \$GI, \$GO.

14.4 DV_CTRL

The following functionalities of DV_CTRL have been removed: 17 (it was reserved), 34, 35.

14.5 Environment variables on the PC

- C4G_IMPORT is changed to C5G_IMPORT
- C4G_GRAMMAR is changed to C5G_GRAMMAR
- C4G_MAKEMESS is changed to C5G_MAKEMESS
- CRC_HOME_CRD is changed to C5G_HOME_CRD
- WINC4G_NOSPLASH is changed to WINC5G_NOSPLASH

14.6 Built-in routines and functions

- DV4_CTRL is now called DV_CTRL.
- DV4_STATE is now called DV_STATE.
- When calling VP2_SCRN_AVAIL Built-in Procedure, the second parameter must now be "tp" and no longer "tp4i" (see also **VP2 - Visual PDL2** Manual).

15. APPENDIX A - CHARACTERS SET

Current chapter lists the ASCII numeric decimal codes and their corresponding hexadecimal codes, character values, and graphics characters, used by the Teach Pendant.

Character values enclosed in parentheses () are ASCII control characters, which are displayed as graphics characters when printed to the screen of the Teach Pendant.



Note that, on a PC (when WINC5G program is active), control characters and graphics characters are displayed according to the currently configured language and characters set.

In the next section (called [Characters Table](#)) a list of all characters, together with the corresponding numeric value in both decimal and hexadecimal representation, follows.

The decimal or hexadecimal numeric value for a character can be used in a STRING value to indicate the corresponding character (for example, '\003 is a graphic character').

15.1 Characters Table

Dec	Hex	Value	Char	Dec	Hex	Value	Char	Dec	Hex	Value	Char
000	0	(NUL)		025	19	(EM)	↓	050	32	2	2
001	1	(SOH)	☺	026	1A	(SUB)	→	051	33	3	3
002	2	(STX)	☻	027	1B	(ESC)	←	052	34	4	4
003	3	(ETX)	♥	028	1C	(FS)	↳	053	35	5	5
004	4	(EOT)	♦	029	1D	(GS)	⊟	054	36	6	6
005	5	(ENQ)	♣	030	1E	(RS)	⊠	055	37	7	7
006	6	(ACK)	♠	031	1F	(US)	⊡	056	38	8	8
007	7	(BEL)	!	032	20	SP		057	39	9	9
008	8	(BS)	₩	033	21	!	!	058	3A	:	:
009	9	(HT)		034	22	"	"	059	3B	;	;
010	A	(LF)		035	23	#	#	060	3C	<	<
011	B	(VT)		036	24	\$	\$	061	3D	=	=
012	C	(FF)		037	25	%	%	062	3E	>	>
013	D	(CR)		038	26	&	&	063	3F	?	?
014	E	(SO)	♪	039	27	'	'	064	40	@	@
015	F	(SI)	☀	040	28	((065	41	A	A
016	10	(DLE)	▶	041	29))	066	42	B	B
017	11	(DC1)	◀	042	2A	*	*	067	43	C	C
018	12	(DC2)	↑↓	043	2B	+	+	068	44	D	D
019	13	(DC3)	!!	044	2C	,	,	069	45	E	E
020	14	(DC4)		045	2D	-	-	070	46	F	F
021	15	(NAK)	§	046	2E	.	.	071	47	G	G
022	16	(SYN)	—	047	2F	/	/	072	48	H	H
023	17	(ETB)	↑↓	048	30	0	0	073	49	I	I
024	18	(CAN)	↑	049	31	1	1	074	4A	J	J

Dec	Hex	Value	Char	Dec	Hex	Value	Char	Dec	Hex	Value	Char
075	4B	K	K	101	65	e	e	127	7F	(DEL)	Δ
076	4C	L	L	102	66	f	f	128	80		Ç
077	4D	M	M	103	67	g	g	129	81		ü
078	4E	N	N	104	68	h	h	130	82		é
079	4F	O	O	105	69	i	i	131	83		â
080	50	P	P	106	6A	j	j	132	84		à
081	51	Q	Q	107	6B	k	k	133	85		å
082	52	R	R	108	6C	l	l	134	86		ç
083	53	S	S	109	6D	m	m	135	87		ê
084	54	T	T	110	6E	n	n	136	88		ë
085	55	U	U	111	6F	o	o	137	89		î
086	56	V	V	112	70	p	p	138	8A		ï
087	57	W	W	113	71	q	q	139	8B		ë
088	58	X	X	114	72	r	r	140	8C		í
089	59	Y	Y	115	73	s	s	141	8D		ï
090	5A	Z	Z	116	74	t	t	142	8E		ä
091	5B	[[117	75	u	u	143	8F		ä
092	5C	\	\	118	76	v	v	144	90		é
093	5D]]	119	77	w	w	145	91		æ
094	5E	^	^	120	78	x	x	146	92		æ
095	5F	-	-	121	79	y	y	147	93		ö
096	60	'	'	122	7A	z	z	148	94		ö
097	61	a	a	123	7B	{	{	149	95		ö
098	62	b	b	124	7C		 	150	96		ö
099	63	c	c	125	7D	}	}	151	97		ö
100	64	d	d	126	7E	~	~	152	98		ÿ

Dec	Hex	Value	Char	Dec	Hex	Value	Char	Dec	Hex	Value	Char
153	99		Ö	179	B3		—	205	CD	=	
154	9A		Ü	180	B4		—	206	CE	¶	
155	9B		¢	181	B5		—	207	CF	£	
156	9C		£	182	B6		—	208	D0	₩	
157	9D		¥	183	B7		—	209	D1	₩	
158	9E		₱	184	B8		—	210	D2	₱	
159	9F		€	185	B9		—	211	D3	£	
160	A0		á	186	BA		—	212	D4	€	
161	A1		í	187	BB		—	213	D5	£	
162	A2		ó	188	BC		—	214	D6	£	
163	A3		ú	189	BD		—	215	D7	£	
164	A4		ñ	190	BE		—	216	D8	£	
165	A5		Ñ	191	BF		—	217	D9	£	
166	A6		à	192	C0		—	218	DA	£	
167	A7		ò	193	C1		—	219	DB	£	
168	A8		¿	194	C2		—	220	DC	£	
169	A9		™	195	C3		—	221	DD	£	
170	AA		™	196	C4		—	222	DE	£	
171	AB		½	197	C5		—	223	DF	£	
172	AC		¼	198	C6		—	224	E0	α	
173	AD		·	199	C7		—	225	E1	ß	
174	AE		«	200	C8		—	226	E2	Γ	
175	AF		»	201	C9		—	227	E3	Π	
176	B0		¤	202	CA		—	228	E4	Σ	
177	B1		¤	203	CB		—	229	E5	σ	
178	B2		¤	204	CC		—	230	E6	μ	



Comau in the World

**COMAU S.p.A.
Headquarters**
Via Rivalta, 30
10095 Grugliasco - TO (Italy)
Tel. +39-011-0049111

Powertrain Machining & Assembly
Via Rivalta, 30-49
10095 Grugliasco - TO (Italy)
Tel. +39-011-0049111
Telefax +39-011-0049688

Body Welding & Assembly
Strada Borgarett, 22
10092 Borgarett di Beinasco - TO (Italy)
Tel. +39-011-0049111
Telefax +39-011-0048672

Robotics & Service
Via Rivalta, 30
10095 Grugliasco - TO (Italy)
Tel. +39-011-0049111
Telefax +39-011-0049866

Engineering, Injection Moulds & Dies
Via Bistagno, 10
10136 Torino (Italy)
Tel. +39-011-0051711
Telefax +39-011-0051882

Comau France S.A.
5-7, rue Albert Einstein
78197 Trappes Cedex (France)
Tel. +33-1-30166100
Telefax +33-1-30166209

Comau Estil
10, Midland Road
Luton, Bedfordshire LU2 0HR (UK)
Tel. +44-1582-817600
Telefax +44-1582-817700

Comau Deutschland GmbH
Monzastrasse 4D
D-63225 Langen (Germany)
Tel. +49-6103-31035-0
Telefax +49-6103-31035-29

German Intec GmbH & Co. KG
Im Riedgrund 1
74078 Heilbronn (Germany)
Tel. +49-7131 28 22-0
Telefax +49-7131 28 22-400

Mecaner S.A.
Calle Aita Gotzon 37
48610 Urduliz - Vizcaya (Spain)
Tel. +34-94-6769100
Telefax +34-94-6769132

Comau Poland Sp. ,Z.O.O.
Ul. Turyńska 100
43-100 Tychy (Poland)
Tel. +48-32-2179404
Telefax +48-32-2179440

Comau Romania S.R.L.
Oradea, 3700 Bihor
Str. Berzei nr.5 Suite E (Romania)
Tel. +40-59-414759
Telefax +40-59-479840

Comau Russia S.R.L.
Ul. Bolshaya Dmitrovka 32/4
107031 Moscow (Russian Federation)
Tel. +7-495-7885265
Telefax +7-495-7885266

Comau SPA Turkiye Bursa Isyeri
Panayır Mah. Buttimis İş Merkezi
C Block Kat 5 no.1494
16250 Osmangazi-Bursa (Turkey)
Tel. +90-0224-2112873
Telefax +90-0224-2112834

Comau Inc.
21000 Telegraph Road
Southfield, MI 48034 (USA)
Tel. +1-248-3538888
Telefax +1-248-3682531

Comau Pico Mexico S. de R.L. de C.V.
Av. Acceso Lotes 12 y 13
Col. Fracc. Ind. El Trébol 2° Secc.
C.P. 54610, Tepotzotlán (Mexico)
Tel. +52-5 8760644
Telefax +52-5 8761837

Comau Canada Inc.
4325 Division Road Unit # 15
Ontario N9A 6J3 (Canada)
Tel. +1-519-9727535
Telefax +1-519-9720809

Comau do Brasil Ind. e Com. Ltda.
Rua Do Paraíso, 148 - 4º Andar
Paraíso - Cep. 04103-000
São Paulo - SP (Brazil)
Tel. +55-11-21262424
Telefax +55-11-32668799

Comau Argentina S.A.
Ruta 9, Km 695
5020 - Ferreyra
Córdoba (Argentina)
Tel. +54-351-4503996
Telefax +54-351-4503909

Comau SA Body Systems (Pty)
Hendrik van Eck Drive
Riverside Industrial Area
Uitenhage 6229 (South Africa)
Tel. +27-41-9953600
Telefax +27-41-9229652

Comau (Shanghai) Automotive Equipment Co., Ltd.
Pudong, Kang Qiao Dong Road Nr. 1300
Block 2 - Kang Qiao
201319 Shanghai (P.R.China)
Tel. +86-21-68139900
Telefax +86-21-68139622

Comau India Pvt. Ltd.
33Km Milestone Pune-Nagar Road
Shikrapur, Pune - 412208 (India)
Tel. +91.2137.678100
Telefax +91.2137.678110

COMAU Robotics services

Repair: repairs.robots@comau.com

Training: training.robots@comau.com

Spare parts: spares.robots@comau.com

Technical service: service.robots@comau.com

comau.com/robotics

Original instructions