

# Comau Robotics Instruction Handbook



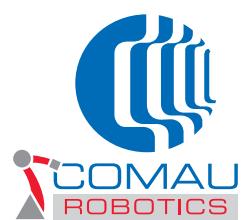
[comau.com/robotics](http://comau.com/robotics)

## PDL2

### Programming Language Manual System Software Rel. 3.1x

**Language Components, Program Structure, Data Representation, Motion Control, Execution Control, I/O Port Arrays, Statements List, Routines, Condition Handlers, Serial Input/Output, Built-in Routines, Predefined Variables, Customizations on the TP, E-mail functionality, Customizing PDL2 Statements in IDE**

CR00757508\_en-01/0208



The information contained in this manual is the property of COMAU S.p.A.

Reproduction of text and illustrations is not permitted without prior written approval by COMAU S.p.A.

COMAU S.p.A. reserves the right to alter product specifications at any time without notice or obligation.

Copyright © 02/2005 by COMAU

# SUMMARY

<b>PREFACE .....</b>	<b>XXVII</b>
Symbols used in the manual .....	XXVII
Reference documents .....	XXVIII
Modification History .....	XXIX
<b>1. GENERAL SAFETY PRECAUTIONS .....</b>	<b>1.1</b>
Responsibilities .....	1.1
Safety Precautions .....	1.2
Purpose .....	1.2
Definitions .....	1.2
Applicability .....	1.3
Operating Modes .....	1.4
<b>2. INTRODUCTION TO PDL2 .....</b>	<b>2.1</b>
Syntax notation .....	2.1
Language components .....	2.3
Character set .....	2.3
Reserved words, Symbols, and Operators .....	2.3
Predefined Identifiers .....	2.5
User-defined Identifiers .....	2.6
Statements .....	2.7
Blank space .....	2.7
Comments .....	2.7
Program structure .....	2.7
Program example .....	2.8
Units of measure .....	2.9
<b>3. DATA REPRESENTATION .....</b>	<b>3.1</b>
Data Types .....	3.1
INTEGER .....	3.2
REAL .....	3.2
BOOLEAN .....	3.3
STRING .....	3.3
ARRAY .....	3.4
RECORD .....	3.5
VECTOR .....	3.6
POSITION .....	3.7
Frames of reference .....	3.10
JOINTPOS .....	3.10

XTNDPOS .....	3.11
NODE .....	3.11
PATH .....	3.12
SEMAPHORE .....	3.13
Declarations .....	3.13
CONSTANT declarations .....	3.13
TYPE declarations .....	3.14
VARIABLE declarations .....	3.16
Shared types, variables and routines .....	3.18
EXPORTED FROM clause .....	3.18
GLOBAL attribute and IMPORT statement .....	3.19
Expressions .....	3.20
Arithmetic Operations .....	3.21
Relational Operations .....	3.22
Logical Operations .....	3.22
Bitwise Operations .....	3.23
VECTOR Operations .....	3.24
POSITION Operations .....	3.24
Data Type conversion .....	3.25
Operator precedence .....	3.25
Assignment Statement .....	3.26
Typecasting .....	3.26
<b>4. MOTION CONTROL .....</b>	<b>4.1</b>
MOVE Statement .....	4.1
ARM Clause .....	4.2
TRAJECTORY Clause .....	4.2
DESTINATION Clause .....	4.3
MOVE TO .....	4.3
VIA Clause .....	4.4
MOVE NEAR .....	4.4
MOVE AWAY .....	4.4
MOVE RELATIVE .....	4.5
MOVE ABOUT .....	4.6
MOVE BY .....	4.6
MOVE FOR .....	4.6
Optional Clauses .....	4.7
ADVANCE Clause .....	4.7
TIL Clause .....	4.8
WITH Clause .....	4.8
SYNCMOVE Clause .....	4.9
Continuous motion (MOVEFLY) .....	4.10
Timing and Synchronization considerations .....	4.11
FLY_CART motion control .....	4.12
Motion along a PATH .....	4.13
ARM Clause .....	4.17
NODE RANGE Clause .....	4.17
Optional Clauses .....	4.18
ADVANCE Clause .....	4.18

WITH Clause .....	4.18
Continuous Motion (MOVEFLY).....	4.20
Stopping and Restarting motions.....	4.20
CANCEL MOTION Statements .....	4.20
LOCK, UNLOCK, and RESUME Statements.....	4.21
SIGNAL SEGMENT Statement .....	4.22
HOLD Statement .....	4.22
ATTACH and DETACH Statements.....	4.22
HAND Statements .....	4.23
<b>5. INPUT/OUTPUT PORT ARRAYS .....</b>	<b>5.1</b>
General .....	5.1
User-defined and Appl-defined Port I/O.....	5.4
\$DIN and \$DOUT.....	5.4
\$IN and \$OUT .....	5.4
\$GIN and \$GOUT .....	5.4
\$AIN and \$AOUT .....	5.5
\$FMI and \$FMO.....	5.5
System-defined Port Arrays.....	5.5
\$SDIN and \$SDOUT .....	5.6
\$FDIN and \$FDOUT .....	5.11
\$HDIN .....	5.15
\$TIMER .....	5.16
Shared Memory Port Arrays .....	5.16
\$BIT .....	5.16
\$WORD .....	5.16
System State After Restart .....	5.17
Cold Start .....	5.18
Power Failure .....	5.18
User-defined Device Access .....	5.18
\$USER_BIT .....	5.19
\$USER_BYTE .....	5.19
\$USER_WORD .....	5.19
\$USER_LONG .....	5.19
\$USER_ADDR .....	5.19
\$USER_LEN .....	5.20
\$PROG_UBIT .....	5.21
\$PROG_UBYTE .....	5.21
\$PROG_UWORD .....	5.21
\$PROG ULONG .....	5.21
\$PROG_UADDR .....	5.21
\$PROG_ULEN .....	5.21
<b>6. EXECUTION CONTROL .....</b>	<b>6.1</b>
Flow Control .....	6.2
IF Statement .....	6.2

SELECT Statement .....	6.4
FOR Statement .....	6.5
WHILE Statement.....	6.7
REPEAT Statement.....	6.8
GOTO Statement .....	6.8
 Program Control.....	6.9
PROG_STATE Built-In Function .....	6.11
ACTIVATE Statement .....	6.11
PAUSE Statement .....	6.11
UNPAUSE Statement.....	6.12
DEACTIVATE Statement.....	6.12
CYCLE and EXIT CYCLE Statements.....	6.13
DELAY Statement .....	6.13
WAIT FOR Statement .....	6.14
BYPASS Statement.....	6.14
 Program Synchronization.....	6.14
Program Scheduling .....	6.16
 <b>7. ROUTINES .....</b>	<b>7.1</b>
Procedures and Functions.....	7.2
Parameters .....	7.2
Declarations.....	7.3
Declaring a Routine .....	7.4
Procedure.....	7.4
Function .....	7.4
Parameter List .....	7.4
Constant and Variable Declarations.....	7.5
Stack Size.....	7.6
Function Return Type.....	7.6
Functions as procedures .....	7.6
RETURN Statement.....	7.6
Shared Routines.....	7.7
 Passing Arguments .....	7.8
Passing By Reference .....	7.8
Passing By Value .....	7.8
Optional parameters.....	7.9
Variable arguments (Varargs) .....	7.10
Argument identifier .....	7.10
 <b>8. CONDITION HANDLERS .....</b>	<b>8.1</b>
Operations .....	8.1
Defining Condition Handlers.....	8.1
FOR ARM Clause.....	8.2
NODISABLE Clause.....	8.3
ATTACH Clause.....	8.3
SCAN Clause .....	8.3
Enabling, Disabling, and Purging .....	8.4

Variable References . . . . .	8.5
Conditions . . . . .	8.5
RELATIONAL States . . . . .	8.6
BOOLEAN State . . . . .	8.6
DIGITAL PORT States . . . . .	8.7
DIGITAL PORT Events . . . . .	8.7
SYSTEM Events . . . . .	8.8
USER Events . . . . .	8.11
ERROR Events . . . . .	8.12
PROGRAM Events . . . . .	8.13
Event on Cancellation of a Suspendable Statement . . . . .	8.13
MOTION Events . . . . .	8.15
Actions . . . . .	8.17
ASSIGNMENT Action . . . . .	8.17
INCREMENT and DECREMENT Action . . . . .	8.17
BUILT-IN Actions . . . . .	8.18
SEMAPHORE Action . . . . .	8.18
MOTION and ARM Actions . . . . .	8.18
ALARM Actions . . . . .	8.19
PROGRAM Actions . . . . .	8.19
CONDITION HANDLER Actions . . . . .	8.19
DETACH Action . . . . .	8.20
PULSE Action . . . . .	8.20
HOLD Action . . . . .	8.20
SIGNAL EVENT Action . . . . .	8.20
ROUTINE CALL Action . . . . .	8.20
Execution Order . . . . .	8.21
<b>9. SERIAL INPUT/OUTPUT . . . . .</b>	<b>9.1</b>
Serial Devices . . . . .	9.1
WINDOW Devices . . . . .	9.1
FILE Devices . . . . .	9.2
PIPE Devices . . . . .	9.3
COMMUNICATION Devices . . . . .	9.3
NULL Device . . . . .	9.5
ATTACH and DETACH Statements . . . . .	9.5
Logical Unit Numbers . . . . .	9.6
OPEN FILE Statement . . . . .	9.6
WITH Clause . . . . .	9.7
CLOSE FILE Statement . . . . .	9.8
READ Statement . . . . .	9.8
Format Specifiers . . . . .	9.9
Power Failure Recovery . . . . .	9.10
WRITE Statement . . . . .	9.10
Output Buffer Flushing . . . . .	9.11
Format Specifiers . . . . .	9.11
Power Failure Recovery . . . . .	9.12
<b>10. STATEMENTS LIST . . . . .</b>	<b>10.1</b>

Introduction . . . . .	10.1
ACTIVATE Statement . . . . .	10.2
ATTACH Statement . . . . .	10.3
BEGIN Statement . . . . .	10.5
BYPASS Statement . . . . .	10.5
CALLS Statement . . . . .	10.6
CANCEL Statement . . . . .	10.7
CLOSE FILE Statement . . . . .	10.8
CLOSE HAND Statement . . . . .	10.9
CONDITION Statement . . . . .	10.9
CONST Statement . . . . .	10.10
CYCLE Statement . . . . .	10.11
DEACTIVATE Statement . . . . .	10.12
DECODE Statement . . . . .	10.13
DELAY Statement . . . . .	10.14
DETACH Statement . . . . .	10.14
DISABLE CONDITION Statement . . . . .	10.15
DISABLE INTERRUPT Statement . . . . .	10.16
ENABLE CONDITION Statement . . . . .	10.17
ENCODE Statement . . . . .	10.18
END Statement . . . . .	10.18
EXIT CYCLE Statement . . . . .	10.19
FOR Statement . . . . .	10.20
GOTO Statement . . . . .	10.21
HOLD Statement . . . . .	10.21
IF Statement . . . . .	10.22
IMPORT Statement . . . . .	10.23
LOCK Statement . . . . .	10.23
MOVE Statement . . . . .	10.24
MOVE ALONG Statement . . . . .	10.26
OPEN FILE Statement . . . . .	10.27
OPEN HAND Statement . . . . .	10.28
PAUSE Statement . . . . .	10.29
PROGRAM Statement . . . . .	10.30
PULSE Statement . . . . .	10.31
PURGE CONDITION Statement . . . . .	10.32

READ Statement . . . . .	10.32
RELAX HAND Statement . . . . .	10.34
REPEAT Statement . . . . .	10.34
RESUME Statement . . . . .	10.35
RETURN Statement . . . . .	10.35
ROUTINE Statement . . . . .	10.36
SELECT Statement . . . . .	10.37
SIGNAL Statement . . . . .	10.38
TYPE Statement . . . . .	10.39
UNLOCK Statement . . . . .	10.40
UNPAUSE Statement . . . . .	10.41
VAR Statement . . . . .	10.41
WAIT Statement . . . . .	10.42
WAIT FOR Statement . . . . .	10.43
WHILE Statement . . . . .	10.43
WRITE Statement . . . . .	10.44
<b>11. BUILT-IN ROUTINES LIST . . . . .</b>	<b>11.1</b>
ABS Built-In Function. . . . .	11.5
ACOS Built-In Function . . . . .	11.5
ACT_POST Built-in Procedure . . . . .	11.5
ARG_COUNT Built-In Function . . . . .	11.6
ARG_GET_VALUE Built-in Procedure . . . . .	11.6
ARG_INFO Built-In Function . . . . .	11.8
ARG_SET_VALUE Built-in Procedure. . . . .	11.9
ARM_COLL_THRS Built-In Procedure . . . . .	11.9
ARM_COOP Built-In Procedure. . . . .	11.10
ARM_GET_NODE Built-In Function . . . . .	11.11
ARM_JNTP Built-In Function. . . . .	11.11
ARM_MOVE_ATVEL Built-in Procedure . . . . .	11.12
ARM_NUM Built-In Function . . . . .	11.12
ARM_POS Built-In Function . . . . .	11.13
ARM_SET_NODE Built-In Procedure . . . . .	11.13
ARM_SOFT Built-In Procedure . . . . .	11.14
ARM_XTND Built-In Function . . . . .	11.15
ARRAY_DIM1 Built-In Function . . . . .	11.15

ARRAY_DIM2 Built-In Function . . . . .	11.15
ASIN Built-In Function . . . . .	11.16
ATAN2 Built-In Function . . . . .	11.16
AUX_COOP Built-In Procedure . . . . .	11.17
AUX_DRIVES Built-In Procedure . . . . .	11.17
AUX_SET Built-In Procedure . . . . .	11.18
BIT_ASSIGN Built-In Procedure . . . . .	11.18
BIT_CLEAR Built-In Procedure . . . . .	11.20
BIT_FLIP Built-In Function . . . . .	11.20
BIT_SET Built-In Procedure . . . . .	11.21
BIT_TEST Built-In Function . . . . .	11.21
CHR Built-In Procedure . . . . .	11.22
CLOCK Built-In Function . . . . .	11.22
COND_ENABLED Built-In Function . . . . .	11.23
COND_ENBL_ALL Built-In Procedure . . . . .	11.23
CONV_SET_OFST Built-In Procedure . . . . .	11.24
COS Built-In Function . . . . .	11.25
DATE Built-In Function . . . . .	11.25
DIR_GET Built-In Function . . . . .	11.26
DIR_SET Built-In Procedure . . . . .	11.26
DRIVEON_DSBL Built-In Procedure . . . . .	11.27
DV4_CTRL Built-In Procedure . . . . .	11.27
DV4_STATE Built-In Function . . . . .	11.32
EOF Built-In Function . . . . .	11.33
ERR_POST Built-In Procedure . . . . .	11.33
ERR_STR Built-In Function v3.11 . . . . .	11.35
ERR_TRAP Built-In Function . . . . .	11.36
ERR_TRAP_OFF Built-In Procedure . . . . .	11.36
ERR_TRAP_ON Built-In Procedure . . . . .	11.37
EXP Built-In Function . . . . .	11.37
FL_BYTES_LEFT Built-In Function . . . . .	11.38
FL_GET_POS Built-In Function . . . . .	11.38
FL_SET_POS Built-In Procedure . . . . .	11.39
FL_STATE Built-In Function . . . . .	11.40
FLOW_MOD_ON Built-In Procedure . . . . .	11.41
FLOW_MOD_OFF Built-In Procedure . . . . .	11.41

HDIN_READ Built-In Procedure .....	11.41
HDIN_SET Built-In Procedure .....	11.42
IP_TO_STR Built-in Function .....	11.43
IS_FLY Built-In Function .....	11.44
JNT Built-In Procedure .....	11.44
JNT_SET_TAR Built-In Procedure .....	11.45
JNTP_TO_POS Built-In Procedure .....	11.46
KEY_LOCK Built-In Procedure .....	11.46
LN Built-In Function .....	11.47
MEM_SPACE Built-In Procedure .....	11.47
NODE_APP Built-In Procedure .....	11.48
NODE_DEL Built-In Procedure .....	11.48
NODE_GET_NAME Built-In Procedure .....	11.49
NODE_INS Built-In Procedure .....	11.49
NODE_SET_NAME Built-In Procedure .....	11.49
ON_JNT_SET Built-In Procedure .....	11.50
ON_JNT_SET_DIG Built-In Procedure .....	11.52
ON_POS Built-In Procedure .....	11.53
ON_POS_SET Built-In Procedure .....	11.55
ON_POS_SET_DIG Built-In Procedure .....	11.55
ON_TRAJ_SET Built-In Procedure .....	11.56
ON_TRAJ_SET_DIG Built-In Procedure .....	11.56
ORD Built-In Function .....	11.56
PATH_GET_NODE Built-In Procedure .....	11.57
PATH_LEN Built-In Function .....	11.58
POS Built-In Function .....	11.58
POS_COMP_IDL Built-In Procedure .....	11.59
POS_CORRECTION Built-In Procedure .....	11.59
POS_FRAME Built-In Function .....	11.60
POS_GET_APPR Built-In Function .....	11.60
POS_GET_CNFG Built-In Function .....	11.60
POS_GET_LOC Built-In Function .....	11.61
POS_GET_NORM Built-In Function .....	11.61
POS_GET_ORNT Built-In Function .....	11.62
POS_GET_RPY Built-In Procedure .....	11.62
POS_IDL_COMP Built-In Procedure .....	11.62

POS_IN_RANGE Built-In Procedure . . . . .	11.63
POS_INV Built-In Function . . . . .	11.64
POS_MIR Built-In Function . . . . .	11.64
POS_SET_APPR Built-In Procedure . . . . .	11.65
POS_SET_CNGF Built-In Procedure . . . . .	11.65
POS_SET_LOC Built-In Procedure . . . . .	11.66
POS_SET_NORM Built-In Procedure . . . . .	11.66
POS_SET_ORNT Built-In Procedure . . . . .	11.67
POS_SET_RPY Built-In Procedure . . . . .	11.67
POS_SHIFT Built-In Procedure . . . . .	11.67
POS_TO_JNTP Built-In Procedure . . . . .	11.68
POS_XTRT Built-In Procedure . . . . .	11.68
PROG_OWNER Built-In Function . . . . .	11.69
PROG_STATE Built-In Function . . . . .	11.69
RANDOM Built-in Function . . . . .	11.71
ROUND Built-In Function . . . . .	11.71
RPLC_GET_IDX Built-In Procedure . . . . .	11.71
SCRN_ADD Built-In Procedure . . . . .	11.72
SCRN_CLEAR Built-In Procedure . . . . .	11.72
SCRN_CREATE Built-In Function . . . . .	11.73
SCRN_DEL Built-In Procedure . . . . .	11.74
SCRN_GET Built-In Function . . . . .	11.74
SCRN_REMOVE Built-In Procedure . . . . .	11.75
SCRN_SET Built-In Procedure . . . . .	11.75
SENSOR_GET_DATA Built-In Procedure . . . . .	11.76
SENSOR_GET_OFST Built-In Procedure . . . . .	11.76
SENSOR_SET_DATA Built-In Procedure . . . . .	11.77
SENSOR_SET_OFST Built-In Procedure . . . . .	11.77
SENSOR_TRK Built-In Procedure . . . . .	11.77
SIN Built-In Function . . . . .	11.78
SQRT Built-In Function . . . . .	11.79
STANDBY Built-In Procedure . . . . .	11.79
STR_CAT Built-In Function . . . . .	11.79
STR_CODING Built-In Function . . . . .	11.80
STR_CONVERT Built-In Function . . . . .	11.80
STR_DEL Built-In Procedure . . . . .	11.81

STR_EDIT Built-In Procedure .....	11.81
STR_GET_INT Built-In Function .....	11.82
STR_GET_REAL Built-In Function .....	11.82
STR_INS Built-In Procedure .....	11.83
STR_LEN Built-In Function .....	11.83
STR_LOC Built-In Function .....	11.84
STR_OVS Built-In Procedure .....	11.84
STR_SET_INT Built-In Procedure.....	11.85
STR_SET_REAL Built-In Procedure .....	11.85
STR_TO_IP Built-In Function .....	11.86
STR_XTRT Built-In Procedure.....	11.86
SYS_ADJUST Built-In Procedure .....	11.87
SYS_CALL Built-In Procedure.....	11.87
SYS_SETUP Built-In Procedure .....	11.88
TABLE_ADD Built-In Procedure .....	11.88
TABLE_DEL Built-In Procedure.....	11.89
TAN Built-In Function.....	11.89
TRUNC Built-In Function .....	11.90
VAR_INFO Built-In Function .....	11.90
VAR_UNINIT Built-In Function.....	11.91
VEC Built-In Function.....	11.91
VOL_SPACE Built-In Procedure .....	11.92
WIN_ATTR Built-In Procedure.....	11.92
WIN_CLEAR Built-In Procedure .....	11.93
WIN_COLOR Built-In Procedure .....	11.94
WIN_CREATE Built-In Procedure .....	11.95
WIN_DEL Built-In Procedure .....	11.96
WIN_DISPLAY Built-In Procedure.....	11.97
WIN_GET_CRSR Built-In Procedure.....	11.97
WIN_LINE Built-In Function.....	11.98
WIN_LOAD Built-In Procedure .....	11.99
WIN_POPUP Built-In Procedure .....	11.100
WIN_REMOVE Built-In Procedure.....	11.101
WIN_SAVE Built-In Procedure.....	11.102
WIN_SEL Built-In Procedure .....	11.103
WIN_SET_CRSR Built-In Procedure.....	11.104

WIN_SIZE Built-In Procedure .....	11.104
WIN_STATE Built-In Function .....	11.105

<b>12. PREDEFINED VARIABLES LIST.....</b>	<b>12.1</b>
Memory Category .....	12.1
Load Category .....	12.1
Minor Category.....	12.2
Data Type .....	12.2
Attributes .....	12.2
Limits .....	12.3
S/W Version.....	12.3
Unparsed .....	12.3
Predefined Variables groups .....	12.3
PLC System Variables .....	12.4
PORT System Variables .....	12.4
PROGRAM STACK System Variables.....	12.4
ARM_DATA System Variables.....	12.5
CRNT_DATA System Variables.....	12.9
DSA_DATA System Variables .....	12.9
MCP_DATA System Variables.....	12.10
FBP_TBL System Variables.....	12.10
WEAVE_TBL System Variables.....	12.10
CONV_TBL System Variables .....	12.10
ON_POS_TBL System Variables .....	12.11
WITH MOVE System Variables .....	12.11
WITH MOVE ALONG System Variables .....	12.12
WITH OPEN FILE System Variables .....	12.13
PATH NODE FIELD System Variables .....	12.14
MISCELLANEOUS System Variables .....	12.14
Alphabetical Listing .....	12.19
\$AIN: Analog input.....	12.30
\$AOUT: Analog output.....	12.30
\$APPL_ID: Application Identifier .....	12.31
\$APPL_NAME: Application Identifiers .....	12.31
\$APPL_OPTIONS: Application Options.....	12.31
\$ARM_ACC_OVR: Arm acceleration override.....	12.32
\$ARM_DATA: Robot arm data.....	12.32
\$ARM_DEC_OVR: Arm deceleration override .....	12.32
\$ARM_DISB: Arm disable flags.....	12.33
\$ARM_LINKED: Enable/disable arm coupling.....	12.33
\$ARM_OVR: Arm override.....	12.33
\$ARM_SENSITIVITY: Arm collision sensitivity .....	12.34

\$ARM_SIMU: Arm simulate flag .....	12.34
\$ARM_SPACE: current Arm Space.....	12.34
\$ARM_SPD_OVR: Arm speed override .....	12.35
\$ARM_USED: Program use of arms .....	12.35
\$ARM_VEL: Arm velocity.....	12.36
\$AUX_BASE: Auxiliary base for a positioner of an arm.....	12.36
\$AUX_KEY: TP4i/WiTP AUX-A and AUX-B keys mapping.....	12.36
\$AUX_MASK: Auxiliary arm mask.....	12.37
\$AUX_OFST: Auxiliary axes offsets .....	12.37
\$AUX_SIK_DRVON_ENBL: Auxiliary axes provided of SIK .....	12.37
\$AUX_SIK_MASK: Auxiliary axes provided of SIK .....	12.38
\$AUX_TYPE: Positioner type .....	12.38
\$AX_CNVRSN: Axes conversion.....	12.38
\$AX_INF: Axes inference.....	12.39
\$AX_LEN: Axes lengths.....	12.40
\$AX_OFST: Axes offsets.....	12.40
\$A_ALONG_1D: Internal arm data .....	12.40
\$A_ALONG_2D: Internal arm data .....	12.40
\$A_AREAL_1D: Internal arm data.....	12.41
\$A_AREAL_2D: Internal arm data.....	12.41
\$B_ASTR_1D_NS: Board string data .....	12.41
\$BACKUP_SET: Default devices.....	12.42
\$BASE: Base of arm .....	12.42
\$BIT: PLC BIT data .....	12.42
\$BOARD_DATA: Board data .....	12.43
\$BOOTLINES: Bootline read-only .....	12.43
\$BREG: Boolean registers - saved .....	12.43
\$BREG_NS: Boolean registers - not saved.....	12.44
\$B_ALONG_1D: Internal arm data .....	12.44
\$B_ALONG_1D_NS: Internal arm data .....	12.44
\$B_NVRAM: NVRAM data of the board .....	12.45
\$C4GOPEN_JNT_MASK: C4G Open Joint arm mask .....	12.45
\$C4GOPEN_MODE: C4G Open modality.....	12.45
\$C4G_RULES: C4G Save & Load rules .....	12.46
\$CAL_DATA: Calibration data .....	12.46
\$CAL_SYS: System calibration position .....	12.46

\$CAL_USER: User calibration position .....	12.47
\$CAUX_POS: Cartesian positioner position .....	12.47
\$CIO_AIN: Configuration for AIN .....	12.48
\$CIO_AOUT: Configuration for AOUT .....	12.48
\$CIO_CAN: Configuration for Can Bus .....	12.48
\$CIO_CROSS: Configuration for I/O cross copying .....	12.49
\$CIO_DIN: Configuration for DIN .....	12.49
\$CIO_DOUT: Configuration for DOUT .....	12.50
\$CIO_FMI: Configuration for \$FMI .....	12.50
\$CIO_FMO: Configuration for \$FMO .....	12.50
\$CIO_GIN: Configuration for GIN .....	12.51
\$CIO_GOUT: Configuration for GOUT .....	12.51
\$CIO_IN_APP: Configuration for IN .....	12.51
\$CIO_OUT_APP: Configuration for OUT .....	12.52
\$CIO_SDIN: Configuration for the system digital inputs .....	12.52
\$CIO_SDOUT: Configuration for the system digital outputs .....	12.52
\$CIO_SYS_CAN: Configuration of the system modules on Can Bus .....	12.53
\$CNFG_CARE: Configuration care .....	12.53
\$CNTRL_CNFG: Controller configuration mode .....	12.53
\$CNTRL_INIT: Controller initialization mode .....	12.54
\$CNTRL_OPTIONS: Controller Options .....	12.55
\$CNTRL_TZ: Controller Time Zone .....	12.56
\$COLL_EFFECT: Collision Effect on the arm status .....	12.56
\$COLL_ENBL: Collision enabling flag .....	12.56
\$COLL_SOFT_PER: Collision compliance percentage .....	12.57
\$COLL_TYPE: Type of collision .....	12.57
\$COND_MASK: PATH segment condition mask .....	12.58
\$COND_MASK_BACK: PATH segment condition mask in backwards .....	12.59
\$CONV_ACC_LIM: Conveyor acceleration limit .....	12.59
\$CONV_BASE: Conveyor base frame .....	12.60
\$CONV_CNFG: Conveyor tracking configuration .....	12.60
\$CONV_DIST: Conveyor shift in micron (mm/1000) .....	12.60
\$CONV_SHIFT: Conveyor shift in mm .....	12.61
\$CONV_SPD: Conveyor speed .....	12.61
\$CONV_SPD_LIM: Conveyor speed limit .....	12.61
\$CONV_TBL: Conveyor tracking table data .....	12.62

\$CONV_TYPE: Element in the Conveyor Table .....	12.62
\$CONV_WIN: Conveyor Windows.....	12.62
\$CONV_ZERO: Conveyor Position Transducer Zero .....	12.63
\$CRNT_DATA: Current Arm data .....	12.63
\$CT_JNT_MASK: Conveyor Joint mask .....	12.63
\$CT_RADIUS: Conveyor radius in mm .....	12.64
\$CT_RES: Conveyor position in motor turns.....	12.64
\$CT_SCC: Conveyor SCC.....	12.64
\$CT_TX_RATE: Transmission rate .....	12.65
\$CUSTOM_ARM_ID: Identifier for the arm.....	12.65
\$CUSTOM_CNTRL_ID: Identifier for the Controller .....	12.65
\$CYCLE: Program cycle count .....	12.66
\$C_ALONG_1D: Internal current arm data .....	12.66
\$C_AREAL_1D: Internal current arm data.....	12.66
\$C_AREAL_2D: Internal current arm data.....	12.67
\$DEPEND_DIRS: Dependancy path.....	12.67
\$DFT_ARM: Default arm .....	12.67
\$DFT_DV: Default devices .....	12.68
\$DFT_LUN: Default LUN number .....	12.68
\$DFT_SPD: Default devices speed.....	12.69
\$DIN: Digital input .....	12.69
\$DNS_DOMAIN: DNS Domain .....	12.69
\$DNS_ORDER: DNS Order.....	12.70
\$DNS_SERVERS: DNS Servers.....	12.70
\$DOUT: Digital output .....	12.70
\$DSA_DATA: DSA data.....	12.71
\$DV_STS: the status of DV4_CNTRL call.....	12.71
\$DV_TOUT: Timeout for asynchronous DV4_CNTRL calls .....	12.71
\$DYN_COLL_FILTER: Dynamic Collision Filter .....	12.72
\$DYN_DELAY: Dynamic model delay .....	12.72
\$DYN_FILTER: Dynamic Filter .....	12.72
\$DYN_FILTER2: Dynamic Filter for dynamic model .....	12.73
\$DYN_GAIN: Dynamic gain in inertia and viscous friction.....	12.73
\$DYN_MODEL: Dynamic Model .....	12.73
\$DYN_WRIST: Dynamic Wrist.....	12.74
\$DYN_WristQs: Dynamic Theta carico .....	12.74

\$D_LONG_1D: Internal DSA data . . . . .	12.74
\$D_AREAL_1D: Internal DSA data . . . . .	12.74
\$D_AXES: Internal DSA data . . . . .	12.75
\$D_CTRL: Internal DSA data. . . . .	12.75
\$D_HDIN_SUSP: DSA_DATA field for HDIN suspend . . . . .	12.75
\$D_MTR: Internal DSA data . . . . .	12.76
\$EMAIL_INT: Email integer configuration . . . . .	12.76
\$EMAIL_STR: Email string configuration. . . . .	12.77
\$ERROR: Last PDL2 Program Error . . . . .	12.77
\$EXE_HELP: Help on Execute command . . . . .	12.77
\$FBP_TBL: Field Bus Table data . . . . .	12.78
\$FB_ADDR: Field Bus ADDR . . . . .	12.78
\$FB_CNFG: Controller fieldbuses configuration mode . . . . .	12.78
\$FB_INIT: Controller fieldbuses initialization mode . . . . .	12.79
\$FB_MA_INIT: Field bus master init . . . . .	12.80
\$FB_MA_SLVS: Field bus master slaves init . . . . .	12.80
\$FB_SLOT: field bus slot. . . . .	12.81
\$FB_SL_INIT: Field bus slave init . . . . .	12.82
\$FB_TYPE: Field bus type. . . . .	12.84
\$FDIN: Functional digital input. . . . .	12.84
\$FDOUT: Functional digital output. . . . .	12.84
\$FL_ADLM: Array of delimiters . . . . .	12.84
\$FL_BINARY: Text or character mode . . . . .	12.85
\$FL_CNFG: Configuration file name . . . . .	12.85
\$FL_COMP: Compensation file name . . . . .	12.85
\$FL_DLMT: Delimiter specification . . . . .	12.86
\$FL_ECHO: Echo characters . . . . .	12.86
\$FL_NUM_CHARS: Number of chars to be read . . . . .	12.86
\$FL_PASSALL: Pass all characters . . . . .	12.87
\$FL_RANDOM: Random file access . . . . .	12.87
\$FL_RDFLUSH: Flush on reading. . . . .	12.87
\$FL_STS: Status of last file operation . . . . .	12.88
\$FL_SWAP: Low or high byte first. . . . .	12.88
\$FLOW_TBL: Flow modulation algorithm table data . . . . .	12.88
\$FLY_DBUG: Cartesian Fly Debug . . . . .	12.89
\$FLY_DIST: Distance in fly motion . . . . .	12.89

\$FLY_PER: Percentage of fly motion . . . . .	12.89
\$FLY_TRAJ: Type of control on cartesian fly . . . . .	12.90
\$FLY_TYPE: Type of fly motion. . . . .	12.90
\$FMI: Flexible Multiple Analog/Digital Inputs. . . . .	12.90
\$FMO: Flexible Multiple Analog/Digital Outputs . . . . .	12.91
\$FOLL_ERR: Following error. . . . .	12.91
\$FUI_DIRS: Installation path . . . . .	12.91
\$FW_ARM: Arm under flow modulation algorithm. . . . .	12.92
\$FW_AXIS: Axis under flow modulation algorithm . . . . .	12.92
\$FW_CNVRSN: Conversion factor in case of Flow modulation algorithm. . . . .	12.92
\$FW_ENBL Flow modulation algorithm enabling indicator. . . . .	12.93
\$FW_FLOW_LIM Flow modulation algorithm flow limit . . . . .	12.93
\$FW_SPD_LIM Flow modulation algorithm speed limits. . . . .	12.93
\$FW_START Delay in flow modulation algorithm application after start . . . . .	12.94
\$FW_VAR: flag defining the variable to be considered when flow modulate is used . . . . .	12.94
\$GEN_OVR: General override. . . . .	12.94
\$GIN: Group input . . . . .	12.95
\$GOUT: Group output . . . . .	12.95
\$GUN: Electrical welding gun . . . . .	12.95
\$HAND_TYPE: Type of hand . . . . .	12.96
\$HDIN: High speed digital input. . . . .	12.96
\$HDIN_SUSP: HDIN Suspend . . . . .	12.96
\$HLD_DEC_PER: Hold deceleration percentage . . . . .	12.97
\$HOME: Arm home position . . . . .	12.97
\$IN: IN digital . . . . .	12.97
\$IPERIOD: Interpolator period. . . . .	12.98
\$IREG: Integer register - saved . . . . .	12.98
\$IREG_NS: Integer registers - not saved . . . . .	12.98
\$JERK: Jerk control values . . . . .	12.98
\$JNT_LIMIT_AREA: Joint limits of the work area . . . . .	12.99
\$JNT_MASK: Joint arm mask . . . . .	12.99
\$JNT_MTURN: Check joint Multi-turn . . . . .	12.99
\$JNT_OVR: joint override . . . . .	12.100
\$JOGL_INCR_DIST: Increment Jog distance . . . . .	12.100
\$JOGL_INCR_ENBL: Jog incremental motion . . . . .	12.100
\$JOGL_INCR_ROT: Rotational jog increment . . . . .	12.101

\$JOG_SPD_OVR: Jog speed override .....	12.101
\$JPAD_DIST: Distance between user and Jpad .....	12.101
\$JPAD_ORNT:TP4i/WiTP Angle setting .....	12.102
\$JPAD_TYPE: TP4i/WiTP Jpad modality rotational or translational .....	12.102
\$JREG: Jointpos registers - saved .....	12.102
\$JREG_NS: Jointpos register - not saved .....	12.103
\$LATCH_CNFG: Latched alarm configuration setting .....	12.103
\$LIN_ACC_LIM: Linear acceleration limit .....	12.103
\$LIN_DEC_LIM: Linear deceleration limit .....	12.104
\$LIN_SPD: Linear speed .....	12.104
\$LIN_SPD_LIM: Linear speed limit .....	12.104
\$LIN_SPD_RT_OVR: Run-time Linear speed override .....	12.104
\$LOG_TO_DSA: Logical to physical DSA relationship .....	12.105
\$LOG_TO_PHY: Logical to physical relationship .....	12.105
\$MAIN_JNTP: PATH node main jointpos destination .....	12.105
\$MAIN_POS: PATH node main position destination .....	12.106
\$MAIN_XTND: PATH node main xtndpos destination .....	12.106
\$MAN_SCALE: Manual scale factor .....	12.106
\$MCP_BOARD: Motion Control Process board .....	12.107
\$MCP_DATA: Motion Control Process data .....	12.107
\$MDM_INT: Modem Configuration .....	12.107
\$MDM_STR: Modem Configuration .....	12.108
\$MOD_ACC_DEC: Modulation of acceleration and deceleration .....	12.108
\$MOD_MASK: Joint mod mask .....	12.108
\$MOVE_STATE: Move state .....	12.108
\$MOVE_TYPE: Type of motion .....	12.109
\$MTR_ACC_TIME: Motor acceleration time .....	12.109
\$MTR_CURR: Motor current .....	12.110
\$MTR_DEC_TIME: Motor deceleration time .....	12.110
\$MTR_SPD_LIM: Motor speed limit .....	12.110
\$M_ALONG_1D: Internal motion control data .....	12.110
\$NET_B: Ethernet Boot Setup .....	12.111
\$NET_B_DIR: Ethernet Boot Setup Directory .....	12.111
\$NET_C_CNFG: Ethernet Client Setting Modes .....	12.111
\$NET_C_DIR: Ethernet Client Setup Default Directory .....	12.112
\$NET_C_HOST: Ethernet Client Setup Remote Host .....	12.112

\$NET_C_PASS: Ethernet Client Setup Password.....	12.112
\$NET_C_USER: Ethernet Client Setup Login Name.....	12.113
\$NET_HOSTNAME: Ethernet network hostnames .....	12.113
\$NET_I_INT: Ethernet Network Information (integers) .....	12.113
\$NET_I_STR: Ethernet Network Information (strings).....	12.114
\$NET_L: Ethernet Local Setup .....	12.114
\$NET_MOUNT: Ethernet network mount .....	12.115
\$NET_Q_STR: Ethernet Remote Interface Information.....	12.115
\$NET_R_STR: Ethernet Remote Interface Setup .....	12.116
\$NET_S_INT: Ethernet Network Server Setup .....	12.116
\$NET_T_HOST: Ethernet Network Time Protocol Host .....	12.117
\$NET_T_INT: Ethernet Network Timer .....	12.117
\$NOLOG_ERROR: Exclude messages from logging .....	12.117
\$NUM_ALOG_FILES: Number of action log files .....	12.118
\$NUM_ARMS: Number of arms.....	12.118
\$NUM_AUX_AXES: Number of auxiliary axes .....	12.118
\$NUM_DEVICES: Number of devices.....	12.118
\$NUM_DSAS: Number of DSAs .....	12.119
\$NUM_JNT_AXES: Number of joint axes .....	12.119
\$NUM_LUNS: Number of LUNs .....	12.119
\$NUM_MB: Number of motion buffers.....	12.120
\$NUM_MB_AHEAD: Number of motion buffers ahead .....	12.120
\$NUM_MCPS: Number of Motion Control Process.....	12.120
\$NUM_PROGS: Number of active programs .....	12.121
\$NUM_SCRNS: Number of screens .....	12.121
\$NUM_TIMERS: Number of timers .....	12.121
\$NUM_VP2_SCRNS: Number of Visual PDL2 screens .....	12.122
\$NUM_WEAVES: Number of weaves (WEAVE_TBL) .....	12.122
\$ODO_METER: average TCP space .....	12.122
\$ON_POS_TBL: ON POS table data.....	12.123
\$OP_JNT: On Pos jointpos .....	12.123
\$OP_JNT_MASK: On Pos Joint Mask.....	12.123
\$OP_POS: On Pos position.....	12.124
\$OP_REACHED: On Posposition reached flag.....	12.124
\$OP_TOL_DIST: On Pos-Jnt Tolerance distance.....	12.124
\$OP_TOL_ORNT: On Pos-Jnt Tolerance Orientation.....	12.125

\$OP_TOOL: The On Pos Tool . . . . .	12.125
\$OP_TOOL_DSBL: On Pos tool disable flag . . . . .	12.125
\$OP_TOOL_RMT: On Pos Remote tool flag . . . . .	12.126
\$OP_UFRAME: The On Pos Uframe . . . . .	12.126
\$ORNT_TYPE: Type of orientation . . . . .	12.126
\$OT_COARSE: On Trajectory indicator . . . . .	12.127
\$OT_JNT: On Trajectory joint position . . . . .	12.127
\$OT_POS: On Trajectory position . . . . .	12.127
\$OT_TOL_DIST: On Trajectory Tolerance distance . . . . .	12.128
\$OT_TOL_ORNT: On Trajectory Orientation . . . . .	12.128
\$OT_TOOL: On Trajectory TOOL position . . . . .	12.128
\$OT_TOOL_RMT: On Trajectory remote tool flag . . . . .	12.129
\$OT_UFRAME: On Trajectory User frame . . . . .	12.129
\$OT_UNINIT: On Trajectory position uninit flag . . . . .	12.129
\$OUT: OUT digital . . . . .	12.129
\$PAR: Nodal motion variable . . . . .	12.130
\$PGOV_ACCURACY: required accuracy in cartesian motions . . . . .	12.131
\$PGOV_MAX_SPD_REDUCTION: Maximum speed scale factor . . . . .	12.132
\$PGOV_ORNT_PER: percentage of orientation . . . . .	12.132
\$POS_LIMIT_AREA: Cartesian limits of work area . . . . .	12.132
\$PPP_INT: PPP Configuration . . . . .	12.133
\$PREG: Position registers - saved . . . . .	12.133
\$PREG_NS: Position registers - not saved . . . . .	12.133
\$PROG_ACC_OVR: Program acceleration override . . . . .	12.134
\$PROG_ARG: Program's activation argument . . . . .	12.134
\$PROG_ARM: Arm of program . . . . .	12.134
\$PROG_CNFG: Program configuration . . . . .	12.135
\$PROG_CONDS: Defined conditions of a program . . . . .	12.135
\$PROG_DEC_OVR: Program deceleration override . . . . .	12.136
\$PROG_NAME: Executing program name . . . . .	12.136
\$PROG_SPD_OVR: Program speed override . . . . .	12.136
\$PROG_UADDR: Address of program user-defined memory access variables . . . . .	12.137
\$PROG_UBIT: Program user-defined bit memory . . . . .	12.137
\$PROG_UBYTE: Program user-defined byte memory . . . . .	12.138
\$PROG_ULEN: Length of program memory access user-defined variables . . . . .	12.138
\$PROG ULONG: Program user-defined long word memory . . . . .	12.138

\$PROG_UWORD: Program user-defined word memory . . . . .	12.139
\$PWR_RCVR: Power failure recovery mode . . . . .	12.139
\$RAD_IDL_QUO: Radian ideal quote . . . . .	12.139
\$RAD_TARG: Radian target . . . . .	12.140
\$RAD_VEL: Radian velocity . . . . .	12.140
\$RBT_CNFG: Robot board configuration . . . . .	12.140
\$RB_FAMILY: Family of the robot arm . . . . .	12.141
\$RB_MODEL: Model of the robot arm . . . . .	12.141
\$RB_NAME: Name of the robot arm . . . . .	12.142
\$RB_STATE: State of the robot arm . . . . .	12.142
\$RB_VARIANT: Variant of the robot arm . . . . .	12.142
\$RCVR_DIST: Distance from the recovery position . . . . .	12.143
\$RCVR_LOCK: Change arm state after recovery . . . . .	12.143
\$RCVR_TYPE: Type of motion recovery . . . . .	12.143
\$READ_TOUT: Timeout on a READ . . . . .	12.144
\$REC_SETUP: RECord key setup . . . . .	12.144
\$REF_ARMS: Reference arms . . . . .	12.145
\$REMOTE: Functionality of the key in remote . . . . .	12.145
\$REM_I_STR: Remote connections Information . . . . .	12.145
\$REM_TUNE: Internal remote connection tuning parameters . . . . .	12.146
\$RESTART: Restart Program . . . . .	12.146
\$RESTART_MODE: Restart mode . . . . .	12.146
\$RESTORE_SET: Default devices . . . . .	12.147
\$ROT_ACC_LIM: Rotational acceleration limit . . . . .	12.147
\$ROT_DEC_LIM: Rotational deceleration limit . . . . .	12.147
\$ROT_SPD: Rotational speed . . . . .	12.148
\$ROT_SPD_LIM: Rotational speed limit . . . . .	12.148
\$RPLC_DATA: Data of PLC resources . . . . .	12.148
\$RPLC_STS: Status of PLC resources . . . . .	12.149
\$RREG: Real registers - saved . . . . .	12.149
\$RREG_NS: Real registers - not saved . . . . .	12.150
\$SAFE_ENBL: Safe speed enabled . . . . .	12.150
\$SDIN: System digital input . . . . .	12.150
\$SDOUT: System digital output . . . . .	12.151
\$SEG_DATA: PATH segment data . . . . .	12.151
\$SEG_FLY: PATH segment fly or not . . . . .	12.151

\$SEG_FLY_DIST: Parameter in segment fly motion.....	12.152
\$SEG_FLY_PER: PATH segment fly percentage .....	12.152
\$SEG_FLY_TRAJ: Type of fly control .....	12.152
\$SEG_FLY_TYPE: PATH segment fly type.....	12.153
\$SEG_OVR: PATH segment override .....	12.153
\$SEG_REF_IDX: PATH segment reference index .....	12.153
\$SEG_STRESS_PER: Percentage of stress required in fly .....	12.154
\$SEG_TERM_TYPE: PATH segment termination type.....	12.154
\$SEG_TOL: PATH segment tolerance .....	12.155
\$SEG_TOOL_IDX: PATH segment tool index.....	12.155
\$SEG_WAIT: PATH segment WAIT .....	12.156
\$SENSOR_CNVRSN: Sensor Conversion Factors.....	12.157
\$SENSOR_ENBL: Sensor Enable.....	12.157
\$SENSOR_GAIN: Sensor Gains .....	12.157
\$SENSOR_OFST_LIM: Sensor Offset Limits .....	12.158
\$SENSOR_TIME: Sensor Time.....	12.158
\$SENSOR_TYPE: Sensor Type .....	12.158
\$SERIAL_NUM: Serial Number.....	12.159
\$SFRAME: Sensor frame of an arm .....	12.159
\$SING_CARE: Singularity care .....	12.160
\$SM4C_STRESS_PER: Maximum Stress allowed in Cartesian SmartMove4 .....	12.160
\$SM4_SAT_SCALE: SmartMove4 saturation thresholds .....	12.160
\$SPD_OPT: Type of speed control .....	12.161
\$SREG: String registers - saved .....	12.161
\$SREG_NS: String registers - not saved.....	12.161
\$STARTUP: Startup program .....	12.162
\$STARTUP_USER: Startup user.....	12.162
\$STRESS_PER: Stress percentage in cartesian fly .....	12.162
\$STRK_END_N: User negative stroke end .....	12.163
\$STRK_END_P: User positive stroke end.....	12.163
\$STRK_END_SYS_N: System stroke ends .....	12.163
\$STRK_END_SYS_P: System stroke ends.....	12.163
\$SWIM_ADDR: SWIM address .....	12.164
\$SWIM_CNGF: SWIM configuration mode .....	12.164
\$SWIM_INIT: SWIM Board initialization parameters .....	12.164
\$SYNC_ARM: Synchronized arm of program .....	12.165

\$SYS_CALL_OUT: Output lun for SYS_CALL .....	12.165
\$SYS_CALL_STS: Status of last SYS_CALL .....	12.166
\$SYS_CALL_TOUT: Timeout for SYS_CALL .....	12.166
\$SYS_ERROR: Last system error.....	12.166
\$SYS_ID: Robot System identifier.....	12.167
\$SYS_INP_MAP: Configuration for system bits in Input on fieldbuses .....	12.167
\$SYS_OUT_MAP: Configuration for system bits in Output on fieldbuses.....	12.167
\$SYS_PARAMS: Robot system identifier .....	12.168
\$SYS_STATE: State of the system .....	12.168
\$TERM_TYPE: Type of motion termination.....	12.169
\$THRD_CEXP: Thread Condition Expression.....	12.170
\$THRD_ERROR: Error of each thread of execution .....	12.170
\$THRD_PARAM: Thread Parameter.....	12.171
\$TIMER: Clock timer .....	12.171
\$TOL_ABT: Tolerance anti-bounce time .....	12.171
\$TOL_COARSE: Tolerance coarse.....	12.172
\$TOL_FINE: Tolerance fine.....	12.172
\$TOL_JNT_COARSE: Tolerance for joints .....	12.172
\$TOL_JNT_FINE: Tolerance for joints.....	12.173
\$TOL_TOUT: Tolerance timeout.....	12.173
\$TOOL: Tool of arm.....	12.173
\$TOOL_CNTR: Tool center of mass of the tool.....	12.174
\$TOOL_FRICTION: Tool Friction .....	12.174
\$TOOL_INERTIA: Tool Inertia.....	12.174
\$TOOL_MASS: Mass of the tool .....	12.175
\$TOOL_RMT: Fixed Tool.....	12.175
\$TOOL_XTREME: Extreme Tool of the Arm.....	12.175
\$TP_ARM: Teach Pendant current arm.....	12.175
\$TP_GEN_INCR: Incremental value for general override.....	12.176
\$TP_MJOG: Type of TP jog motion.....	12.176
\$TP_ORNT: Orientation for jog motion .....	12.176
\$TP_SYNC_ARM: Teach Pendant's synchronized arms .....	12.177
\$TUNE: Internal tuning parameters .....	12.177
\$TURN_CARE: Turn care .....	12.178
\$TX_RATE: Transmission rate .....	12.179
\$UFRAME: User frame of an arm .....	12.179

\$USER_ADDR: Address of user-defined variables . . . . .	12.179
\$USER_BIT: User-defined bit memory . . . . .	12.180
\$USER_BYTE: User-defined byte memory . . . . .	12.180
\$USER_LEN: Length of User-defined variables . . . . .	12.180
\$USER_LONG: User-defined long word memory . . . . .	12.181
\$USER_WORD: User-defined word memory . . . . .	12.181
\$VERSION: Software version . . . . .	12.181
\$VP2_SCRN_ID: Executing program VP2 Screen Identifier . . . . .	12.182
\$VP2_TOUT: Timeout value for asynchronous VP2 requests . . . . .	12.182
\$VP2_TUNE: Visual PDL2 tuning parameters . . . . .	12.182
\$WEAVE_MODALITY: Weave modality . . . . .	12.183
\$WEAVE_MODALITY_NOMOT: Weave modality (only for no arm motion) . . . . .	12.183
\$WEAVE_NUM: Weave table number . . . . .	12.183
\$WEAVE_NUM_NOMOT: Weave table number (only for no arm motion) . . . . .	12.184
\$WEAVE_PHASE: Index of the Weaving Phase . . . . .	12.184
\$WEAVE_TBL: Weave table data . . . . .	12.184
\$WEAVE_TYPE: Weave type . . . . .	12.185
\$WEAVE_TYPE_NOMOT: Weave type (only for no arm motion) . . . . .	12.185
\$WFR_IOTOUT: Timeout on a WAIT FOR when IO simulated . . . . .	12.185
\$WFR_TOUT: Timeout on a WAIT FOR . . . . .	12.186
\$WORD: PLC WORD data . . . . .	12.186
\$WRITE_TOUT: Timeout on a WRITE . . . . .	12.186
\$WV_AMP_PER: Weave amplitude percentage . . . . .	12.187
\$WV_CNTR_DWL: Weave center dwell . . . . .	12.187
\$WV_END_DWL: Weave end dwell . . . . .	12.187
\$WV_LEFT_AMP: Weave left amplitude . . . . .	12.188
\$WV_LEFT_DWL: Weave left dwell . . . . .	12.188
\$WV_LENGTH_WAVE: Wave length . . . . .	12.188
\$WV_ONE_CYCLE: Weave one cycle . . . . .	12.189
\$WV_PLANE: Weave plane angle . . . . .	12.189
\$WV_RIGHT_AMP: Weave right amplitude . . . . .	12.189
\$WV_RIGHT_DWL: Weave right dwell . . . . .	12.190
\$WV_SMOOTH: Weave smooth enabled . . . . .	12.190
\$WV_SPD_PROFILE: Weave speed profile enabled . . . . .	12.190
\$WV_TRV_SPD: Weave transverse speed . . . . .	12.191
\$WV_TRV_SPD_PHASE: Weave transverse speed phase . . . . .	12.191

\$XREG: Xtndpos registers - saved . . . . .	12.191
\$XREG_NS: Xtndpos registers - not saved . . . . .	12.192
<b>13. POWER FAILURE RECOVERY . . . . .</b>	<b>13.1</b>
<b>14. TRANSITION FROM C3G TO C4G CONTROLLER . . . . .</b>	<b>14.1</b>
Introduction . . . . .	14.1
System parameters values . . . . .	14.1
Terminology . . . . .	14.2
Old programs compatibility with the new Controller Unit . . . . .	14.2
Devices and Directories . . . . .	14.2
System Variables . . . . .	14.3
File <\$SYS_ID>.C4G . . . . .	14.3
Removed System Variables . . . . .	14.3
Variables which have been renamed . . . . .	14.4
\$FDIN, \$FDOUT, \$SDIN, \$SDOUT . . . . .	14.5
BIT referenced System Variables . . . . .	14.5
Array element referenced System Variables . . . . .	14.5
Miscellaneous . . . . .	14.5
Predefined constants . . . . .	14.6
Built-in routines and functions . . . . .	14.6
Communication Device Control . . . . .	14.6
SYS_CALL . . . . .	14.6
WIN_LOAD . . . . .	14.7
Built-ins that have been removed . . . . .	14.7
Open File Statement . . . . .	14.7
Conditions . . . . .	14.7
Swapping to a foreign language . . . . .	14.7
System menu Commands . . . . .	14.8
Removed System Menu Commands . . . . .	14.8
‘Configure’ commands . . . . .	14.8
‘Filer’ commands . . . . .	14.9
‘Input/Output’ commands . . . . .	14.9
Utility Comm Mount / Dism . . . . .	14.9
PLC commands . . . . .	14.9
Replaced Commands . . . . .	14.9
Compressed files . . . . .	14.9
Error and action Logging . . . . .	14.10
PC interface . . . . .	14.10
Fieldbuses configuration . . . . .	14.10
MMUX (motor multiplexer) . . . . .	14.10

<b>15. APPENDIX A -</b>	<b>CHARACTERS SET .....</b>	<b>15.1</b>
Characters Table .....	15.2	
<b>16. APPENDIX B -</b>	<b>CUSTOMIZATIONS ON THE TP .....</b>	<b>16.1</b>
Introduction .....	16.1	
User table creation from DATA environment .....	16.1	
Properties .....	16.2	
Table (global) Properties .....	16.2	
Field Properties .....	16.6	
Field of a POSITION Properties .....	16.8	
Variable <type>_signal .....	16.9	
Example program for a table creation .....	16.10	
Handling TP4i/WiTP right menu .....	16.11	
XML Tag Parameters .....	16.12	
Softkey Pressure and Release events .....	16.13	
Example of XML configuration file .....	16.14	
Example of right menu configuration .....	16.15	
Customizing the PDL2 Statements insertion, in IDE environment .....	16.16	
Adding User Instructions to the PDL2 menu .....	16.17	
Adding a Statement .....	16.17	
Adding a Routine .....	16.18	
Adding a Built-in Routine .....	16.18	
Creating a "virtual" Numeric Keypad to insert User Instructions .....	16.20	
<b>17. APPENDIX C -</b>	<b>E-MAIL FUNCTIONALITY .....</b>	<b>17.1</b>
Introduction .....	17.1	
Configuration of SMTP and POP3 clients .....	17.1	
Sending/receiving e-mails on C4G Controller .....	17.2	
"email" program .....	17.2	
Sending PDL2 commands via e-mail .....	17.5	

# 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 **C4G Control Unit**.

The complete set of manuals for the **C4G** consists of:

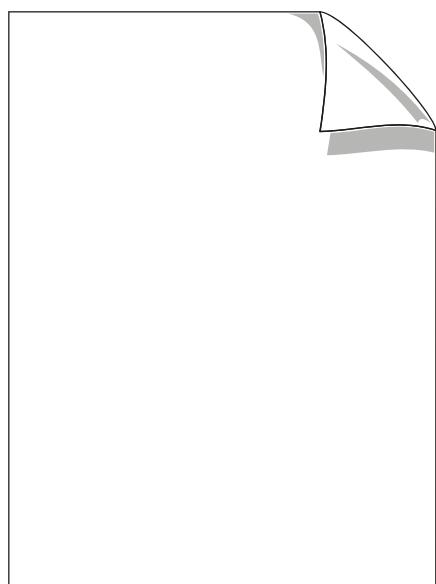
Comau	C4G Control Unit	<ul style="list-style-type: none"><li>– Technical Specifications</li><li>– Transport and installation</li><li>– Guide to integration, safeties, I/O and communications</li><li>– Use of Control Unit.</li></ul>
-------	------------------	---

These manuals are to be integrated with the following documents:

Comau	Robot	<ul style="list-style-type: none"><li>– Technical Specifications</li><li>– Transport and installation</li><li>– Maintenance</li></ul>
	Programming	<ul style="list-style-type: none"><li>– PDL2 Programming Language Manual</li><li>– VP2 - Visual PDL2</li><li>– Motion programming</li><li>– EZ PDL2 - Programming environment</li></ul>
	Applications	<ul style="list-style-type: none"><li>– According to the required type of application.</li></ul>
Altersys	PLC programming	<ul style="list-style-type: none"><li>– <u>ISaGRAF Workbench</u></li></ul>

## Modification History

- The **VP2** chapter has been removed from the current Manual, and a new **VP2 - Visual PDL2** Manual has been created.
- The following Built-ins have been added:
  - POS\_COMP\_IDL
  - POS\_IDL\_COMP.
- In System Software version 3.12, the following paragraphs have been added:
  - [User table creation from DATA environment](#)
  - [Customizing the PDL2 Statements insertion, in IDE environment.](#)



# 1. GENERAL SAFETY PRECAUTIONS

---

## 1.1 Responsibilities

- The system integrator is responsible for ensuring that the [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 and Control System](#), by tampering with circuits, components or software, or the use of spare parts that are not originals or that have not been defined as equivalent by COMAU Robotics & Service
- The application of these Safety Precautions is the responsibility of the persons assigned to direct / supervise the activities indicated in the [Applicability](#) section, They 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 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 and Control System**

The Robot and Control System consists of all the functions that cover: Control Unit, robot, hand held programming unit and any 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;
- Stop distances (threshold values)
- 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 C4G Control Unit manages internally the main safety interlocks (gates, enabling pushbuttons, etc.). Connect the C4G 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 ESK 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 ESK 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 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 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.**

### Stop distances (threshold values)

- As for the stop distance threshold values for each robot type, please turn to the COMAU Robotics & Service Dept.
- Example: Considering the robot in automatic mode, in conditions of maximum extension, maximum load and maximum speed, when the stop pushbutton is pressed (red mushroom head pushbutton on WiTP) an NJ 370-2.7 Robot will stop completely in approx. 85° of motion, equivalent to approx. 3000 mm displacement measured on the TCP flange. Under these conditions indicated, the stoppage time of the NJ 370-2.7 Robot is 1.5 seconds.
- Considering the robot in programming mode (T1), when the stop pushbutton is pressed (red mushroom head pushbutton on WiTP) an NJ 370-2.7 Robot will stop completely in approx. 0.5 seconds.

### 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.

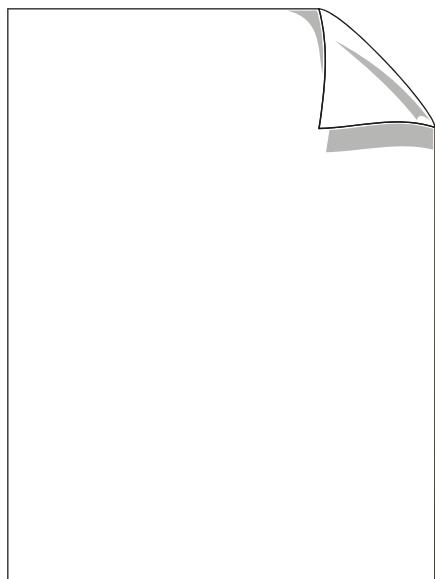


**After replacement of the ESK module, check on the new module that the setting of the stop time on selector switches SW1 and SW2 is consistent with the type of Robot connected to the Control Unit.**

- 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 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.



## 2. INTRODUCTION TO PDL2

---

The current chapter contains information about the following topics:

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

PDL2 is a Pascal-like language with special features for programming robotic applications. Programming means developing a list of instructions that the controller executes to perform a particular application. Programs can include instructions to do the following:

- move robotic arms;
- send and receive information;
- control the order in which instructions are executed;
- check for errors or other special conditions.

PDL2 is used to program the Comau C4G controller. PDL2 offers more structure than the original PDL, making it possible to program more complex applications with greater control and fewer errors. A complete PDL2 programming environment is included with the language.

The C4G controller can execute multiple programs simultaneously to handle all aspects of an application. A single controller can also direct multiple robotic arms and related equipment.

PDL2 programs are divided into two categories, depending on the holdable/non-holdable attribute.

- holdable programs (indicated by the HOLD attribute) are controlled by START and HOLD. 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. 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.

### 2.1 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 item3 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.

## 2.2 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

### 2.2.1 Character set

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

**Tab. 2.1 - PDL2 Character Set**

<b>Letters:</b>	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
<b>Digits:</b>	0 1 2 3 4 5 6 7 8 9
<b>Symbols:</b>	@ < > = / * + - _ , ; . # \$ [ ] % { } \ : ! ( )
<b>Special Characters:</b>	blank (space), tab

PDL2 does not distinguish between uppercase and lowercase letters.

### 2.2.2 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
CYCLE	CYCLES	DEACTIVATE	DEACTIVATES

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

DECODE	DELAY	DETACH	DISABLE
DISTANCE	DIV	DO	DOWNT0
DV4_CNTRL	ELSE	ENABLE	ENCODE
END	ENDCONDITION	ENDFOR	ENDIF
ENDMOVE	ENDNODEDEF	ENDOPEN	ENDRECORD
ENDSELECT	ENDTRY	ENDWHILE	ERRORCLASS
ERRONUM	EVENT	EXECS	EXIT
EXPORTED	EZ	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	[
]	{	}	(
)	<	>	<=
>=	<>	=	+
-	*	/	**
,	.	..	:
::	:=	;	#
@	+=	-=	

## 2.2.3 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. [Tab. 2.3 - Predefined Constants](#) lists the predefined constants.

**Tab. 2.3 - Predefined Constants**

AC_ABORT	AC_CALL_CRT	AC_CALL_TP	BASE
CIRCULAR	COARSE	COLL_LOW	COLL_MEDIUM
COLL_HIGH	COLL_MANUAL	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_READ	CONV_2READ	CONV_1READ
CONV_OFF	CONV_1ON	CONV_1ON_2READ	CONV_2ON
CONV_1READ_2ON	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
EUL_WORLD	FALSE	FINE	FLY_AUTO
FLY_CART	FLY_FROM	FLY_NORM	FLY_PASS
FLY_TOL	JNT_COARSE	JNT_FINE	JOINT
JPAD_LIN	JPAD_ROT	LANG_EN	LANG_FR
LANG_DE	LANG_IT	LANG_PO	LANG_SP
LANG_TR	LANG_ZH	LINEAR	LUN_CRT
LUN_NULL	LUN_TP	MAXINT	MININT
NOSETTLE	OFF	ON	ON_MV
PDV_CRT	PDV_TP	RPY_WORLD	RS_TRAJ
RS_WORLD	SCRN_ALARM	SCRN_APPL	SCRN_DATA
SCRN_FILE	SCRN_IDE	SCRN_IO	SCRN_LOGIN
SCRN_MOTION	SCRN_PROG	SCRN_SERVICE	SCRN_SETUP
SCRN_TPINT	SCRN_CLR_CHR	SCRN_CLR_DEL	SCRN_CLR_Rem
SCRN_EDIT	SCRN_SYS	SCRN_USER	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_LWR

**Tab. 2.3 - Predefined Constants**

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_GREEN	WIN_MAGENTA	WIN_RED	WIN_REVERSE
WIN_SCROLL	WIN_WHITE	WIN_WRAP	WIN_YELLOW
WRIST_JNT	WIN_YELLOW		

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.

## 2.2.4 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, which are also used to name the file in which the program is stored, cannot exceed eight characters.

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



**Program identifiers, which are also used to name the file in which the program is stored, cannot exceed eight characters.**

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.

## 2.3 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.

### 2.3.1 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.

### 2.3.2 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.

## 2.4 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.

## 2.5 Program example

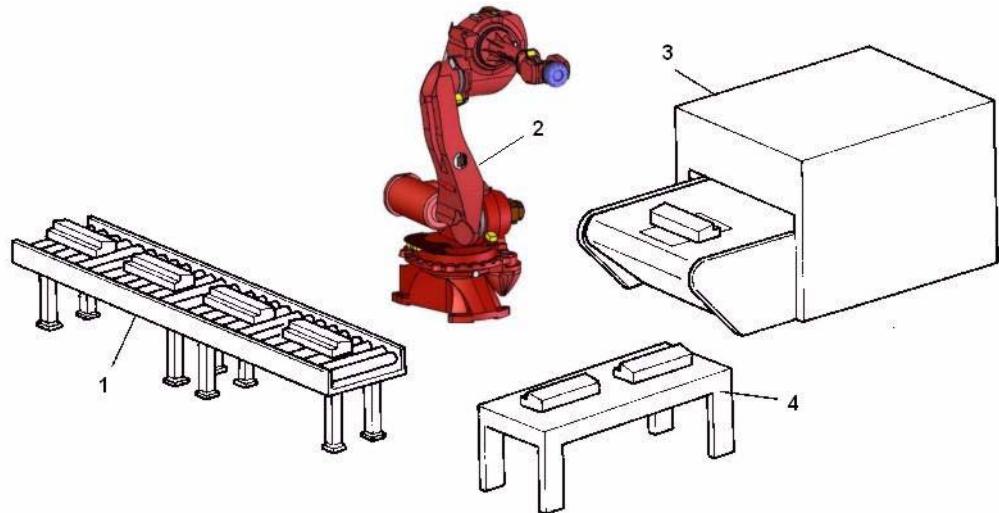
The following PDL2 program transfers parts from a feeder to a work table or a discard bin, as shown in [Fig. 2.1 - Pack Program Work Area](#). Digital input signals indicate when the feeder is ready and whether or not the part should be discarded.

```

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**



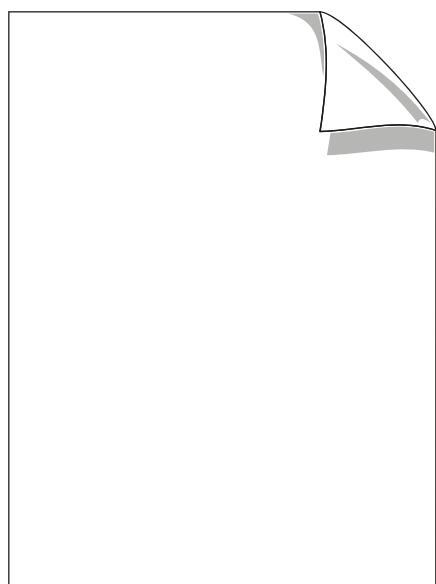
1. Feeder
2. Robot
3. Discard Bin
4. Table

## 2.6 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 ( $^{\circ}$ )
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 3-17](#)). 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 0b 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 ASCII characters, treated as a single unit of data. Single quotes mark the beginning and the end of a string value. For example:

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

In addition to printable characters, strings can contain control sequences. A 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: ~
```

All nonprintable characters in a STRING value not represented in the above format, are replaced with blanks. Note that, when working in PROGRAM EDIT or in MEMORY DEBUG, control sequences for printable characters are viewed in the corresponding printable character itself.

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

```
Literal: 'Single quote'' '
Value:      Single quote '
```

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

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 11-3](#)).

In the case of using strings of UNICODE characters, double quotes mark the beginning and the end of the string value.

Examples:

"this is a UNICODE string value"

"程序机构可以进行编辑并示教开发程序 "

In addition to printable characters, strings can contain control sequences. The control sequence for UNICODE consists of \u followed by the requested UNICODE code.

For example:

UNICODE Literal: "UNICODE 8A89 is \u0x8A89"

UNICODE value: UNICODE 8A89 is 誉

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

To produce either the backslash () or the double quotes ("") characters in a STRING literal, use two in a row. For example:

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

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



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

**UNICODE characters are internally represented by means of 2 bytes, whereas ASCII characters are represented by means of 1 byte only.**

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 11-3](#)).

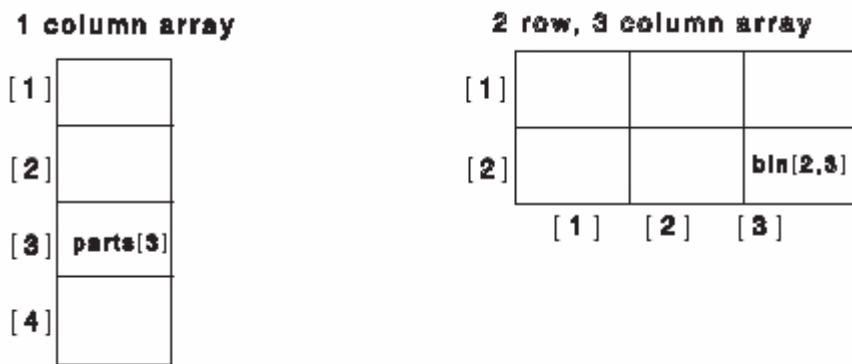
### 3.1.5 ARRAY

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 3-14](#) and [par. 3.2.4.2 GLOBAL attribute and IMPORT statement on page 3-19](#))

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.

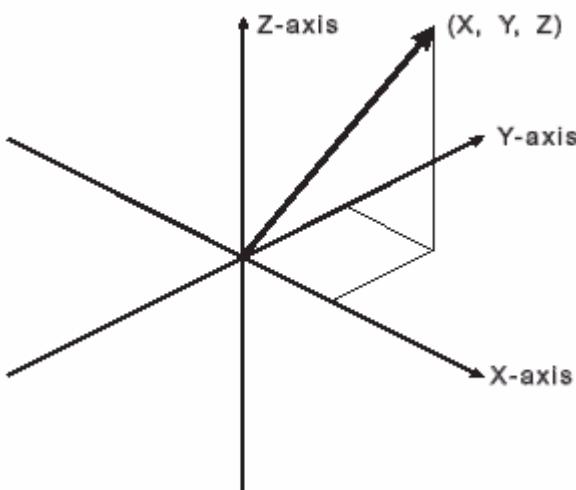
PDL2 can perform the following operations on VECTOR data:

- 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 3-10](#) 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**

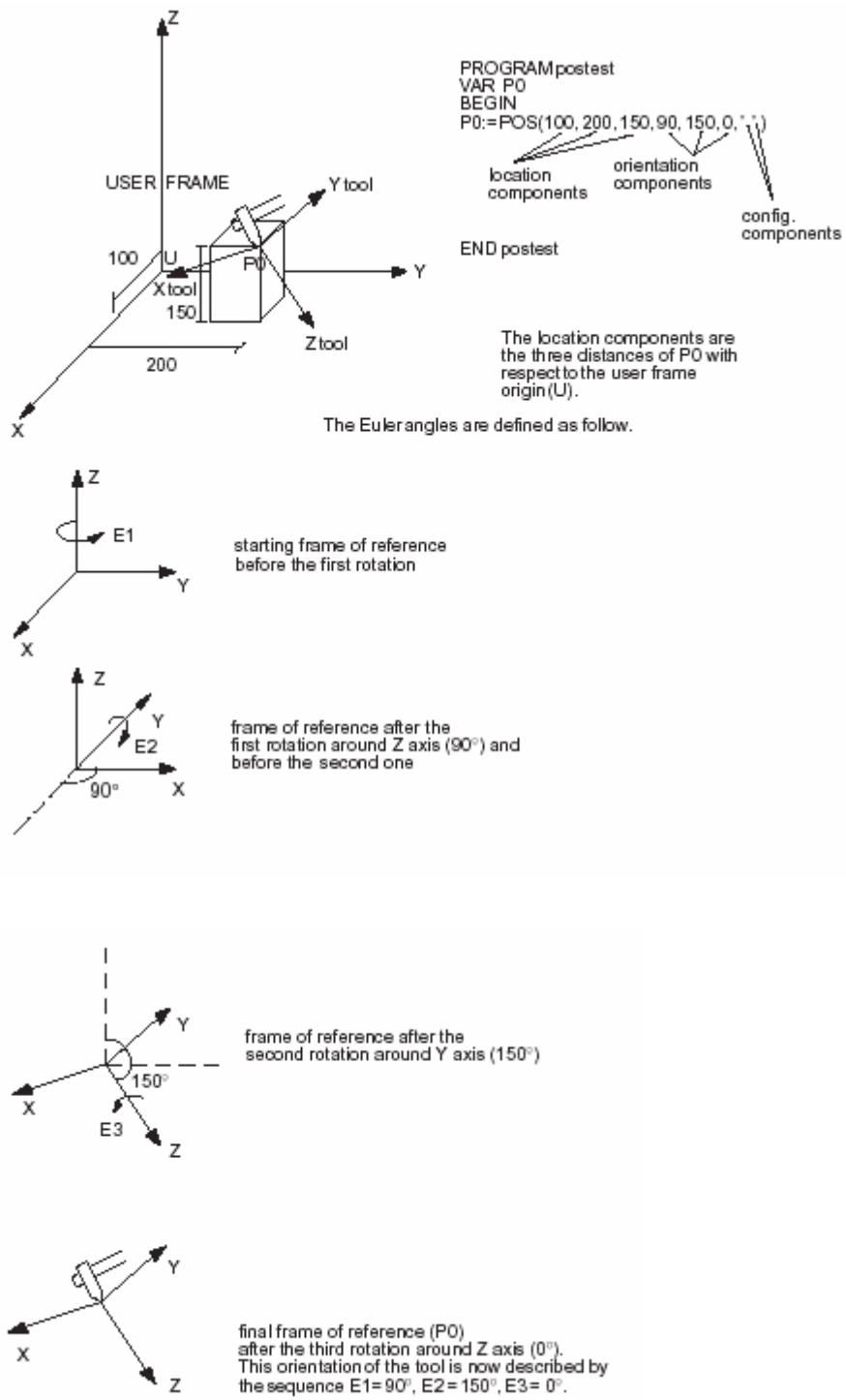
<b>NORMAL CONFIGURATION</b>	
<b>Turn Flags</b>	<b>Robot Axis</b>
T 1	ax 4
T 2	ax 6
T 3	ax 3
T 4	ax 5

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

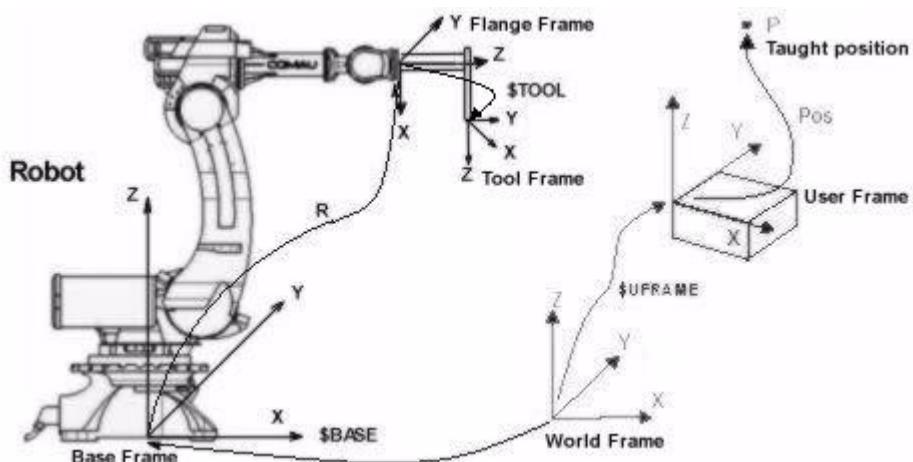
```

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**



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 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 3-14](#) and [par. 3.2.4.2 GLOBAL attribute and IMPORT statement on page 3-19](#))

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. 10.11 CONST Statement on page 10-10](#)). A constant declaration establishes a constant identifier with

two attributes; a name and an unchanging value. A constant's data type 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. 10.45 TYPE Statement on page 10-39](#)). 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 3-18](#) 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 *Use of C4G Controller Unit* 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. 10.48 VAR Statement on page 10-41](#)). 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 3-4)
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 3-1](#) 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 *Use of C4G Controller Unit* 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 3-18](#) 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 *Use of C4G Controller Unit* 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 [C4G Control Unit Use](#) manual).

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 -- x and ary are open for other programs to use -- y is owned by program b</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 -- y is open for other programs to use -- x and ary are owned by program a</pre>
--	--



**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. 10.27 IMPORT Statement on page 10-23](#)).

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.

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:

<i>count</i> + 1	-- arithmetic
VEC( <i>a</i> , <i>b</i> , <i>c</i> ) * 2	-- arithmetic
<i>count</i> >= <i>total</i>	-- relational
<i>flag_1</i> AND <i>flag_2</i>	-- 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:	I	I	R	I	B	I	—	—	I	I	I
INTEGER	R	R	R	—	B	—	—	—	—	—	—
Operators:	+, -	*	/	DIV, MOD	=,<> <,<= >,>=	AND, NOT, OR, XOR			ROR ROL SHR SHL	**	+= -=
REAL	R	R	R	—	B	—	—	—	—	—	—
INTEGER-REAL	R	R	R	—	B	—	—	—	—	R	—
REAL-INTEGER	—	—	—	—	B	B	—	—	—	—	—
BOOLEAN	—	—	—	—	B	—	—	—	—	—	—
STRING	—	V	V	—	—	—	—	—	—	—	—
INTEGER-VECTOR	—	V	V	—	—	—	—	—	—	—	—
REAL-VECTOR	V	—	—	—	B <sup>(1)</sup>	—	V	R	—	—	—
VECTOR	—	—	—	—	—	—	—	—	P	—	—
POSITION	—	—	—	—	—	—	—	—	V	—	—
POSITION-VECTOR	I	I	R	I	B	I	—	—	I	I	I

(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)

<b>/</b>	REAL or VECTOR division
<b>MOD</b>	INTEGER modulus (remainder)
<b>**</b>	INTEGER or REAL exponentiation
<b>+ =</b>	INTEGER increment
<b>- =</b>	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:

<b>&lt;</b>	less than
<b>&gt;</b>	greater than
<b>=</b>	equal
<b>&lt;=</b>	less than or equal
<b>&gt;=</b>	greater than or equal
<b>&lt;&gt;</b>	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.

<b>AND</b>	TRUE if both operands are TRUE
<b>OR</b>	TRUE if at least one of the operands is TRUE
<b>XOR</b>	TRUE if only one operand is TRUE
<b>NOT</b>	inverse of the BOOLEAN operand

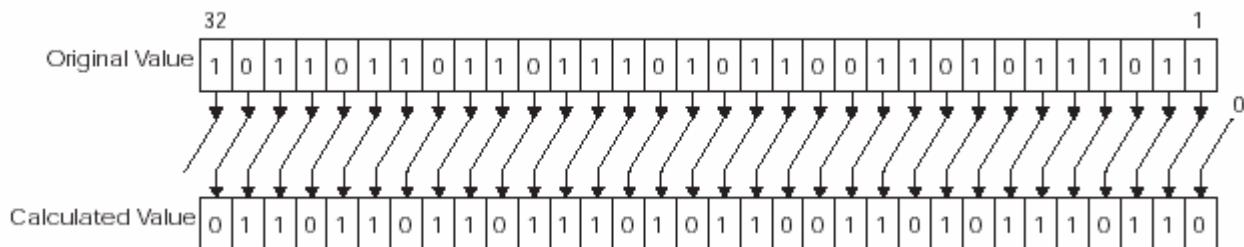
### 3.3.4 Bitwise Operations

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**



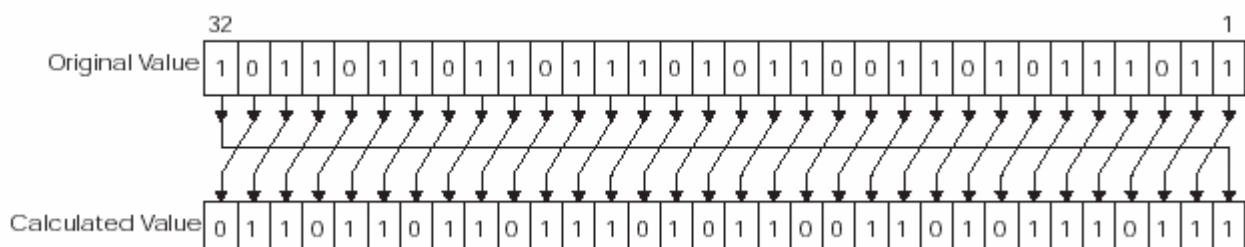
[Fig. 3.7](#) shows the result of the following rotate operation:

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



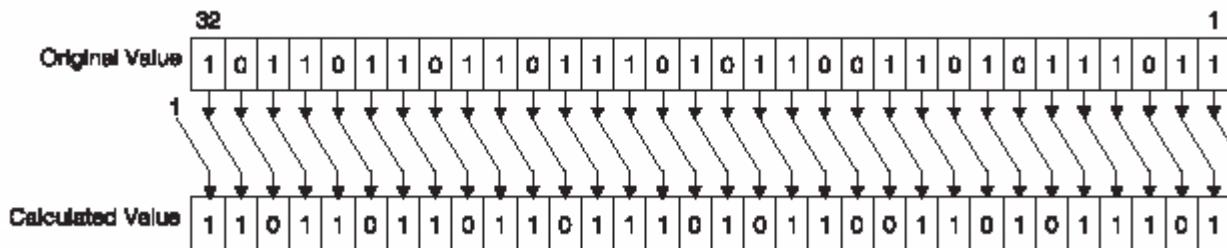
**NOTE** that the Shift Left operation might cause the variable to become UNINIT: at run time, an INTEGER PDL2 variable will assume the value of UNINIT (uninitialized) if it becomes greater than MAXINT.

**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.

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

$$\text{VEC}(2, 5, 7) \# \text{VEC}(3, 9, 1) = \text{VEC}(-58, 19, 3)$$

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:

$$\text{VEC}(7.0, 6.5, 13.4) @ \text{VEC}(1.3, 5.2, 0.0) = 42.9$$

### 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)
```

### 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
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)

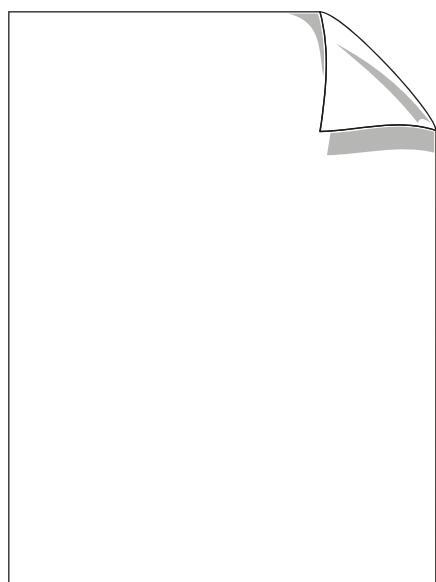
CONDITION[5] :
    -- if real_var values 5.5 and int_var 0x3f9999A
    -- the condition will trigger
    WHEN (INTEGER)real_var > int_var DO
        real_var := (REAL)3
    WHEN $AOUT[13] < (INTEGER) real_var DO
        int_var := (INTEGER)5.6
    WHEN real_var > (REAL)int_var DO
        $AOUT[7] :=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

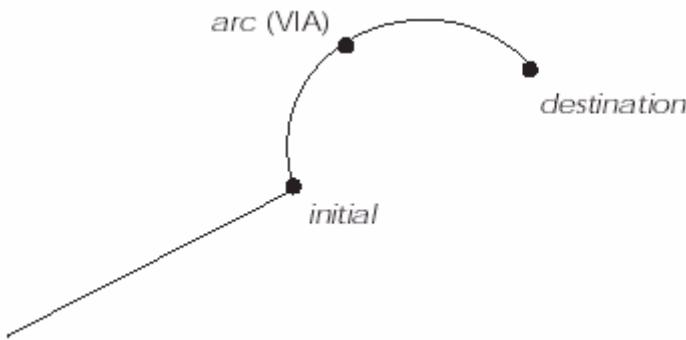
```
MOVE TO { , , gamma, delta}
-- only joints 3 and 4 are moved
```

#### 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

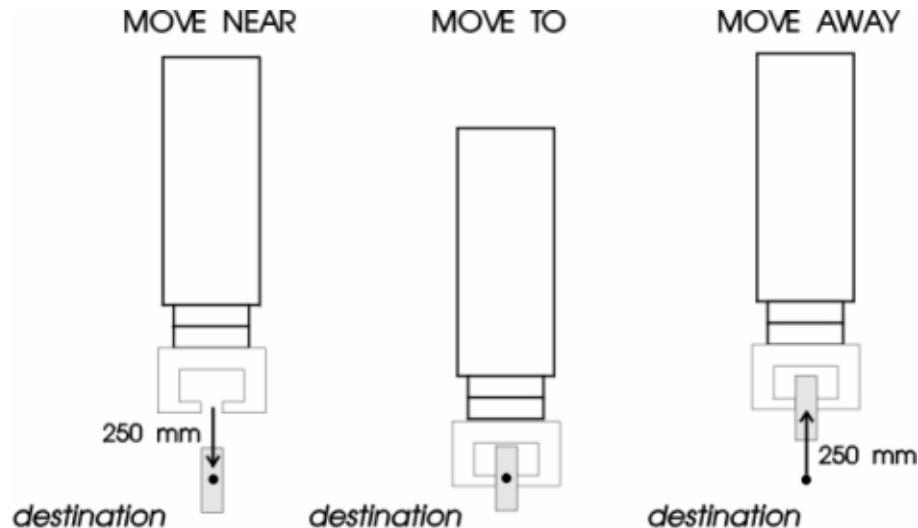
MOVE AWAY allows the programmer to specify a destination along the tool approach vector that is a specified distance away from the current position. The distance, specified as a real expression, is measured in millimeters along the negative tool approach vector.

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.

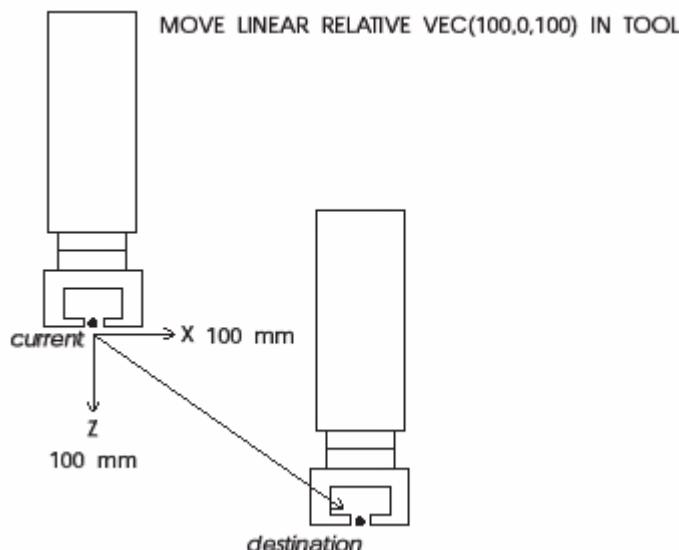
For example:

```
MOVE 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:

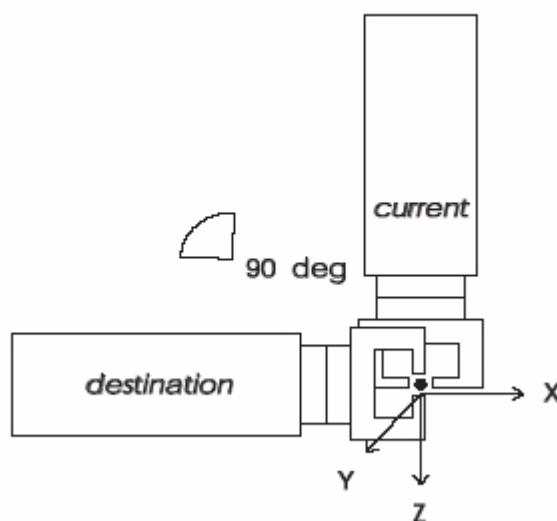
```
MOVE ABOUT VEC(0, 100, 0) BY 90 IN TOOL
```

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`

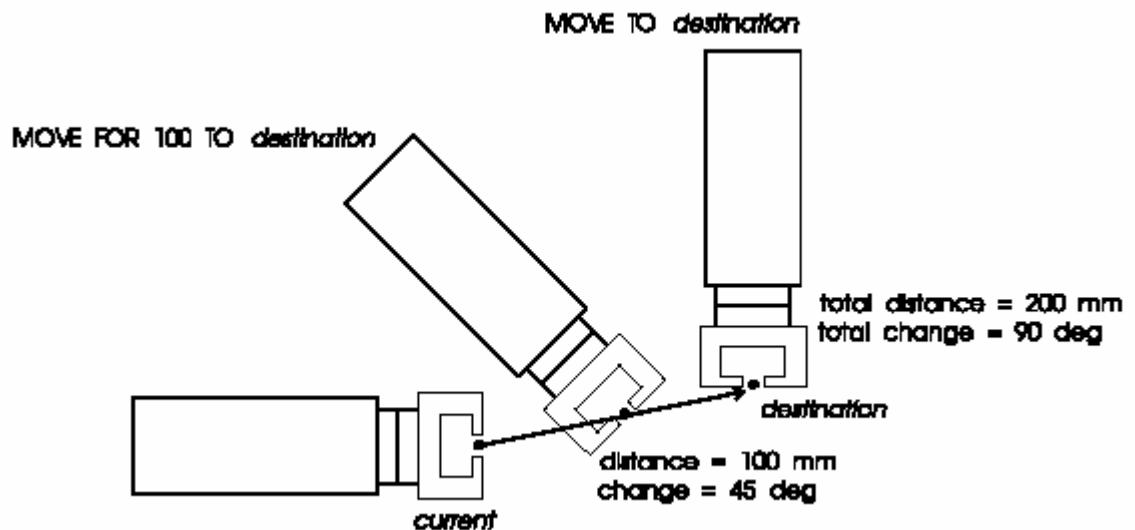
The *distance* is represented by a real expression. If the value is positive, the move is measured toward the destination. If the value is negative, the move is measured in the opposite direction. The distance is measured in millimeters.

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 NEAR slot BY 250
OPEN HAND 1      -- executed after move near motion is completed
MOVE NEAR slot BY 250 ADVANCE
OPEN HAND 1      -- executed while move near motion is in progress
MOVE TO slot      -- executed after move near motion is completed
```

#### 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 affects only the motion caused by the 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
\$SFRAAME	\$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
```

The WITH clause associated to the \$PAR predefined variable implements a particular way of writing MOVE statements, called *move with \$PAR*. Refer to the \$PAR description present in [Chap. Predefined Variables List](#) for further details.

#### 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.



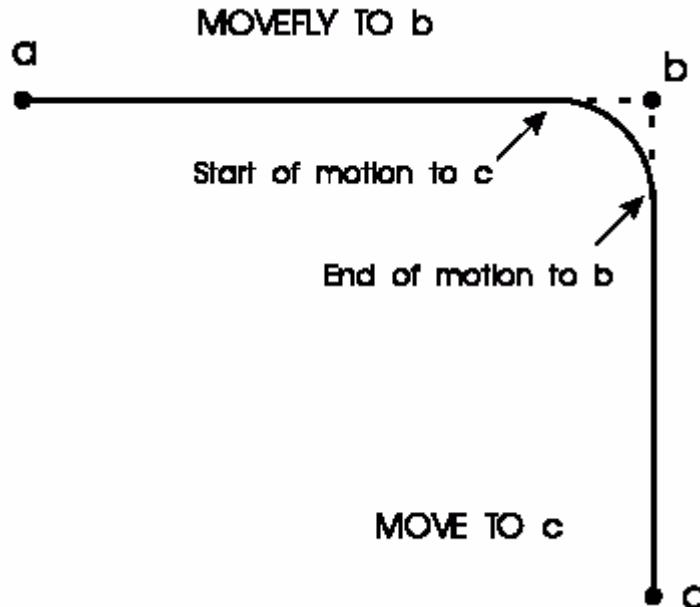
**It is not necessary for the two trajectories of the fly motion to have the same Base or Frame, but it is necessary to have the same Tool!**

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 predefine variable \$FLY\_PER only effects the arm speed if the predefine 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 decleration 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 \$CFG\_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

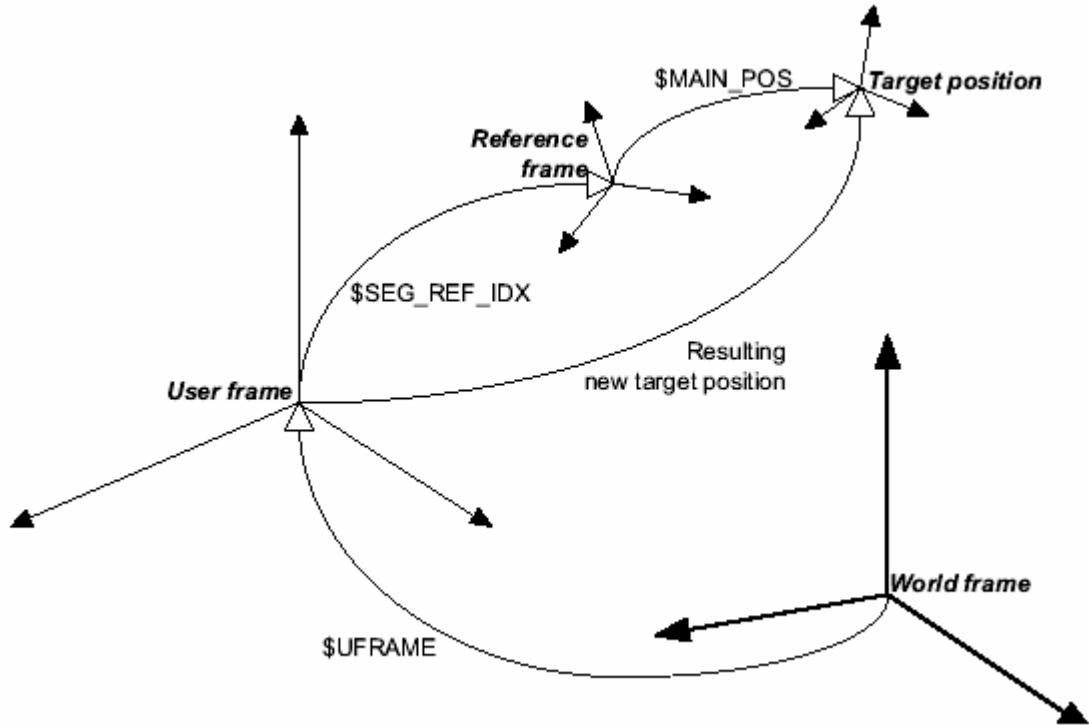
```

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
ENDCONDITION
.
.
.
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. INPUT/OUTPUT PORT ARRAYS

---

## 5.1 General

Current chapter explains the following types of port arrays used in PDL2:

- digital, group, flexible, and analog port arrays, configured by the user to accommodate specific application I/O ([par. 5.2 User-defined and Appl-defined Port I/O on page 5-4](#));
- system-defined port arrays, internally mapped for system devices such as operator devices, arms, and timers ([par. 5.3 System-defined Port Arrays on page 5-5](#));
- shared memory port arrays used by PDL2 programs to communicate one another ([par. 5.4 Shared Memory Port Arrays on page 5-16](#));
- user-defined port arrays used for user-defined device access ([par. 5.6 User-defined Device Access on page 5-18](#)).

[Tab. 5.1 - Port Arrays](#) lists the I/O-related port arrays used by PDL2.

**Tab. 5.1 - Port Arrays**

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
\$GIN	INTEGER	group input	user-def
\$GOUT	INTEGER	group output	user-def
\$AIN	INTEGER	analog input	user-def
\$AOUT	INTEGER	analog output	user-def
\$SDIN	BOOLEAN	digital input	sys-def
\$FMI	INTEGER	flexible multiple input	appl-def
\$FMO	INTEGER	flexible multiple output	appl-def
\$SDOUT	BOOLEAN	digital output	sys-def
\$FDIN	BOOLEAN	digital input	sys-def
\$FDOUT	BOOLEAN	digital output	sys-def
\$HDIN	BOOLEAN	digital input	sys-def
\$TIMER	INTEGER	system timers	sys-def
\$BIT	BOOLEAN	shared memory	PDL2-PLC
\$WORD	INTEGER	shared memory	PDL2-PLC
\$USER_BIT	BOOLEAN	user-def var	user-def
\$USER_BYTE	INTEGER	user-def var	user-def
\$USER_LONG	INTEGER	user-def var	user-def

**Tab. 5.1 - Port Arrays (Continued)**

Identifier	Data Type	Description	Category
\$USER_WORD	INTEGER	user-def var	user-def
\$PROG_UBIT	BOOLEAN	user-def var	user-def
\$PROG_UBYTE	INTEGER	user-def var	user-def
\$PROG_UWORD	INTEGER	user-def var	user-def
\$PROG ULONG	INTEGER	user-def var	user-def

Depending on the type of array, each item in a port array represents a particular input or output signal or group of signals, or a shared memory location. For user-defined port arrays, 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 := $GIN[3]
SELECT body_type OF
CASE(1):
    $GOUT[1] := four_door
CASE(2):
    $GOUT[2] := two_door
CASE(3):
    $GOUT[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 array. 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 port arrays and the system-defined \$FDOOUT port array.

When the Controller is in PROGR state, additional checks are performed before allowing an output to be set. The additional rules for setting the outputs of a digital, analog or group port are listed below. If any of these conditions are met, the output cannot be set:

- by an active program;

- when executing a statement that sets an output from the PC keyboard (WinC4G Program is active) while the Teach Pendant is out of the cabinet. An exception to this rule exists if a PDL2 program has bit 3 of \$PROG\_CNFG set to 0; in this case all output settings are allowed. (\$PROG\_CNFG is a program system variable that defaults to the current value of \$CNTRL\_CNFG when the program is activated);
- when executing under the PROGRAM EDIT or the MEMORY DEBUG Commands from the PC video/keyboard (WinC4G Program) and the Teach Pendant is out of the cabinet.

In addition, these outputs cannot be set in AUTO mode with the Teach Pendant out of the cabinet.

**Tab. 5.2 - Protection on I/O**

	Execute		SetOutputF		PDL2Prog		PE/MD		
\$PROG_CNFG	0 1 1		0 1 1		0 1 1		0 1 1		
\$CNTRL_CNFG		0 1 1		0 1 1		0 1 1		0 1 1	
(1)*	D D		D D		A I D A I D		D D		\$DOUT Sys_call
(2)*	A A		A A		A I D A I D		A A		\$DOUT Sys_call
(3)	A A		A A		A A		A A		\$DOUT Sys_call

[Tab. 5.2 - Protection on I/O](#) shows the cases in which the setting of these outputs is allowed or disallowed when operating in PROGR state. Here follow some notes useful for understanding [Tab. 5.2 - Protection on I/O](#):

- ‘A’ indicates the operation is allowed, ‘D’ means disallowed. The position of A or D depends on the variable whose value allows or disallows the setting of the I/O. For example, if ‘A’ is below \$CNTRL\_CNFG, this means that the operation described at that line is based on the value of that variable. If only a ‘A’ or ‘D’ is present, the operation is always allowed/disallowed independent of the value of the variable. It remains in the column to indicate that the variable can determine the permission of the operation in other circumstances. When ‘A’ or ‘D’ is in the middle of the box, it means that the operation is independent from the value of \$PROG\_CNFG and \$CNTRL\_CNFG. When the space is blank, it means that the protection is not related to the corresponding variable (\$PROG\_CNFG or \$CTRL\_CNFG);
- (1): Case of Operations Panel switch turned to T1, TP out of cabinet. Operations performed from the PC keyboard (WinC4G tool active);
- (2): Case of Operations Panel switch turned to T1, TP out of cabinet. Operations performed from TP;
- (3): Case of Operations Panel switch turned to T1, TP on cabinet. Operations performed from the PC keyboard (WinC4G tool active);
- (\*): This situation occurs when, starting with the TP out of the cabinet in PROGR state, the Operations Panel switch is turned to REMOTE without putting the TP on the cabinet. This situation is reported in bit 7 of the \$SYS\_STATE system variable;

- **Execute** is the command available at the system menu for executing a PDL2 statement at the command level;
- **SetOutputF** is an abbreviation of the SetOutputForce/Unforce commands and identifies an example of all the commands that act on outputs in the system;
- **PDL2 Prog** refers to the state when the action on an output is performed from a PDL2 program;
- **PE/MD** indicates PROGRAM EDIT and MEMORY DEBUG. This column illustrates what happens when a PDL2 statement that acts on an output is executed when operating in the environments shown;
- **\$DOUT** indicates a PDL2 statement acting on an output (i.e. `$DOUT(XX) := ON`);
- **Sys\_call** indicates a SYS\_CALL of a command that acts on an output, for example `SYS_CALL('SOFD', '17', 'T')`, and not another SYS\_CALL such as `SYS_CALL('E','$DOUT(17) := T')`.

## 5.2 User-defined and Appl-defined Port I/O

Digital, group, flexible, and analog port arrays are configured by the user to accommodate specific application I/O.

A detailed description follows about:

- **\$DIN and \$DOUT**
- **\$IN and \$OUT**
- **\$GIN and \$GOUT**
- **\$AIN and \$AOUT**
- **\$FMI and \$FMO**

### 5.2.1 \$DIN and \$DOUT

\$DIN and \$DOUT port arrays 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 port arrays, reserved to application programs.

### 5.2.3 \$GIN and \$GOUT

\$GINs and \$GOUTs allow the user to group discrete I/O points (up to 16) and read or write them as if they were an analog signal (or binary).

There are a maximum of 32 \$GIN and 32 \$GOUT.

For example, the user could set `$GOUT[1]` to 127, and this would set outputs corresponding to the binary pattern 000000001111111.

PDL2 treats them as INTEGER values. For example:

```
IF $GIN[n] > 8
.
.
$GOUT[n] := 4
```

The group of signals can have a size of 1 to 16 bits, where each bit corresponds to an input or output line. Unused bits are treated as zero. For example, where x is ON and o is OFF in an 8-bit group:

Group of 8 Signals	Integer Value
o o o o o o o x	1
o o o o o o x o	2
o o o o o x o x	5
o o o o x o x x	11

Any association between digital and group I/O is done at the system level as part of the user-defined I/O configuration. An input or output port that is associated with a group input or output can be accessed using the \$DIN or \$DOUT or \$IN or \$OUT port array.

Each \$GIN/\$GOUT can be composed by a maximum of two groups of the same port type. For example, \$GOUT[3] can include \$DOUT[3..7] and \$DOUT[12..14].

## 5.2.4 \$AIN and \$AOUT

\$AIN and \$AOUT port arrays 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.

## 5.2.5 \$FMI and \$FMO

\$FMI and \$FMO are Flexible Multiple port arrays, reserved to application programs. PDL2 treats them as INTEGER values.

# 5.3 System-defined Port Arrays

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

A detailed description follows about:

- [\\$SDIN and \\$SDOUT](#)
- [\\$FDIN and \\$FDOUT](#)
- [\\$HDIN](#)
- [\\$TIMER](#)

### 5.3.1 \$SDIN and \$SDOUT

\$SDIN and \$SDOUT port arrays 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 \$SDIN signal meanings:

\$SDIN	Meaning
REMOTE signals used with System CAN Module	
1	reserved
2	TP inserted (°)
3 - 4	reserved
5	AUTO selector position
6	T1 selector position
7	reserved
8	T2 selector position
9	DRIVE ON from remote
10	Not DRIVE OFF from remote
11	Start from remote
12	Not Hold from remote
13	U1 from remote
14	U2 from remote
15	U3 from remote
16	U4 from remote
17..21	reserved
22	REMOTE selector position
23-32	reserved (*)
System CAN signals	
33	Emergency Stop channel 1
34	Emergency Stop channel 2
35	Enabling Device channel 1
36	Enabling Device channel 2
37	Safety Gates channel 1
38	Safety Gates channel 2
39	General Stop channel 1
40	General Stop channel 2
41	State T1
42	State T2
43	State AUTO
44	State REMOTE
45	T1 & T2 state relay
46	AUTO & REMOTE state relay

47	Contactors are low
48	Power contactors channel 1
49	Power contactors channel 2
50	Series of safety contactors channel 1
51	Series of safety contactors channel 2
52	Power contactors command channel 1
53	Power contactors command channel 2
54	Application contactors are low
55	Monitoring Emergency Stop button
56	TP inserted (°)
57-96	reserved (*)
Signals for optional IEAK feature	
Available on any Fieldbus	
97	MOTOR ON 1st rotating table
98	MOTOR ON 2nd rotating table
99	MOTOR ON 3rd rotating table
100	MOTOR ON 4th rotating table
101	No MOTOR OFF 1st rotating table
102	No MOTOR OFF 2nd rotating table
103	No MOTOR OFF 3rd rotating table
104	No MOTOR OFF 4th rotating table
Available on user module CAN only	
105	No TIMER MOTOR OFF 1st rotating table
106	No TIMER MOTOR OFF 2nd rotating table
107	No TIMER MOTOR OFF 3rd rotating table
108	No TIMER MOTOR OFF 4th rotating table
109	SIK ON 1st rotating table
110	SIK ON 2nd rotating table
111	SIK ON 3rd rotating table
112	SIK ON 4th rotating table
113-144	reserved
Remote Control Fieldbus signals	
145	DRIVE ON
146	No DRIVE OFF
147	Start
148	Not hold
149	U1
150	U2
151	U3
152	U4
153	Cancel alarm

154	Safety speed
155..160	Application dependent
161-162	Output permission channel 1 and 2
163-164	RPU electronics OK channel 1 and 2
165	Output of internal relay
166	Pairing Switch
167-168	Expansion 1 and 2 presence
169	Expansion Relay Output
170	Expansion Relay Emergency Stop
171	Relay Stop tp wireless
172	Expansion Relay Stop
173	Emergency Stop enable delay
174	Auto Stop enable delay
175	General Stop enable delay
176	Docking Station presence
177-180	Pairing state code
181-200	Reserved
201	Last telegram crc ok
202	Timestamp checks ok
203	ESM TX telegram requests
204	TP is paired
205	Reserved
206	Command received from PDO1
207	Pair switch state
208	Safe block output checked
209-216	Reserved
217	Signal latch output latched
218-222	Reserved
223	Automatic state
224	Programming state
225-226	Wireless Enbl device channel 1 and 2
227-228	Wireless Stop TP channel 1 and 2
229-230	Wireless Power Enbl channel 1 and 2
231	EnDev AND AutoSt signals
232	Delayed signals
233-234	Reserved
235	Wireless stop relay cmd
236	Power cutting relay cmd
237	Emcy stop Ext dup relay cmd
238	Stop exp output relay
239	Power contactor K101 relay cmd
240	Output timesetting mode not 0
241-244	State of pairing

245	Output ok diagnostic signal
246	Hardware ok diagnostic signal
247	Software ok diagnostic signal
248	All ok diagnostic signal
249-256	Raw data received from RXPDO2

**Note:** "Application dependent" bits are only used after a call, by a PDL2 program, to ON\_POS\_SET\_DIG (followed by ON\_POS) or ON\_TRAJ\_SET\_DIG built-in.

(\*) When Sik devices are present in the System, in order to handle the SIK feature, these signals are reserved by the System for I/O configuration of the Sik device. For further details about Sik device and how to use it, refer to [AUX\\_DRIVES Built-In Procedure](#) in current manual, [Use of C4G Controller manual - IO\\_TOOLS Program - IO\\_INST Program - SETUP Page](#) chapters, [Integration guidelines Safeties, I/O, Communications](#) manual.

(^) Note that if the ESM safety module is present, the information related to \$SDIN[56] is "Wired TP" (1 = TP4i, 0 = WiTP)

Here follows a list of \$SDOUT signal meanings.

\$SDOUT	Meaning
REMOTE signals used with System CAN Module	
1	Internal DRIVE ON command
2	WiTP pairing available (if ESM safety module is present)
3	reserved
4	Not Alarm to remote
5..8	reserved
9	DRIVE ON / Not OFF to remote
10	Start / Not Hold in running to remote
11	Not AUTO / REMOTE to remote
12	Teach Enable (DRIVE ON + T1 or T2) / Not Disable to remote
13	U1 to remote (LED)
14	U2 to remote (LED)
15	U3 to remote (LED)
16	U4 to remote (LED)
17	T2
18	Panel Start
19	Panel reset
20	Local emergency stop
21	System I/O simulated
22-32	reserved (*)
Signals for optional IEAK feature	
Available on any Fieldbus	
33	MOTOR ON/OFF STS 1st rotating table
34	MOTOR ON/OFF STS 2nd rotating table

35	MOTOR ON/OFF STS 3rd rotating table
36	MOTOR ON/OFF STS 4th rotating table
37..40	reserved
Available on user module CAN only	
41	RESTORE 1st rotating table
42	RESTORE 2nd rotating table
43	RESTORE 3rd rotating table
44	RESTORE 4th rotating table
45..48	reserved
49-96	reserved (*)
System State information	
97	Standby
98	Jog active
99	Holdable programs running
100	Em/Auto/Gen stop alarm active
101	winC4G on PC connected
102	reserved
103	TP disconnected in PROGR state
104	State selector on T2
105	Power Failure recovery in progress
106	Teach Pendant on cabinet
107	Enabling device activated
108 - 109	reserved
110	State selector on REMOTE
111	State selector on AUTO
112	State selector on T1
113	Hold request to remote
114	Hold button latched on TP
115	reserved
116	DRIVE OFF to remote
117	DRIVE OFF button latched on
118	reserved
119	TP connected
120	Alarm due to high severity error
121	System in REMOTE
122	Start button pressed
123	System in alarm
124	System in PROGR / AUTO-T
125	System in AUTO / REMOTE
126	System in Hold
127	Drives status
128	reserved

Remote Status Fieldbus signals	
129	Reference position 1
130	Reference position 2
131	Reference position 3
132	Reference position 4
133	Reference position 5
134	Reference position 6
135	Reference position 7
136	Reference position 8
137	Collision alarm DSA1
138	Collision enabled DSA1
139	Collision alarm DSA2
140	Collision enabled DSA2
141..144	reserved
145	Not alarm
146	DRIVE ON / Not DRIVE OFF
147	Start / Not hold in running
148	REMOTE
149	Teach Enable (DRIVE ON + PROGR or AUTO-T) / Not Disable
150	U1
151	U2
152	U3
153	U4
154	reserved
155	reserved
156	PROGR mode
157	AUTO-T mode
158	Robot in AUTO / REMOTE
159	Ready for work
160	Heart bit

(\*) When Sik devices are present in the System, in order to handle the SIK feature, these signals are reserved by the System for I/O configuration of the Sik device. For further details about Sik device and how to use it, refer to [AUX\\_DRIVES Built-In Procedure](#) in current manual, [Use of C4G Controller manual - IO\\_INST Program - IO\\_TOOL Program - SETUP Page](#) chapters, [Integration guidelines Safeties, I/O, Communications manual - Axes expansion](#) chapter.

### 5.3.2 \$FDIN and \$FDOUT

The \$FDIN and \$FDOUT port arrays 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 (WinC4G 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 TP4i/WiTP or the menus of IDE, PROGRAM EDIT and MEMORY DEBUG 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 IDE, I/O Page, PROGRAM EDIT, MEMORY DEBUG, and EZ environments. These are marked with an asterisk in the "Element" column.

**Tab. 5.3 - \$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]	state selector on T2
\$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
\$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 (**)

**Tab. 5.3 - \$FDIN (Continued)**

\$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
\$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 (**)

**Tab. 5.3 - \$FDIN (Continued)**

\$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]	EZ softkey (**)
\$FDIN[106]	A1 softkey (**)
\$FDIN[107]	A2 softkey (**)
\$FDIN[108]	RESET softkey (**)
\$FDIN[109..128]	reserved
\$FDIN[129]	F1 on PC keyboard (WinC4G Program active)
\$FDIN[130]	F2 on PC keyboard (WinC4G Program active)
\$FDIN[131]	F3 on PC keyboard (WinC4G Program active)
\$FDIN[132]	F4 on PC keyboard (WinC4G Program active)
\$FDIN[133]	F5 on PC keyboard (WinC4G Program active)
\$FDIN[134]	F6 on PC keyboard (WinC4G Program active)
\$FDIN[135]	F7 on PC keyboard (WinC4G Program active)
\$FDIN[136]	F8 on PC keyboard (WinC4G 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 TP4i/WiTP Teach Pendant, these \$FDIN/\$FDOUT detect variations only if TPINT Page or Virtual Keyboard are active.

**Tab. 5.4 - \$FDOUT**

Element	Associated Input
\$FDOUT[1..16]	reserved
\$FDOUT[17]	DRIVE ON led
\$FDOUT[18..20]	reserved
\$FDOUT[21]	U1
\$FDOUT[22]	U2
\$FDOUT[23]	U3
\$FDOUT[24]	U4
\$FDOUT[25]	ALARM led. When an alarm occurs, this led is lighted

**Tab. 5.4 - \$FDOUT (Continued)**

\$FDOUT[26]	T1 led for HAND 1 of the selected arm. It is lighted when the HAND is closed
\$FDOUT[27]	T2 led for HAND 2 of the selected arm. It is lighted when the HAND is closed
\$FDOUT[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 (**)
\$FDOUT[29]	enables the Save Screen function on the TP display
\$FDOUT[30]	EZ STEP DISB led
\$FDOUT[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 (**)
\$FDOUT[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 (**)
\$FDOUT[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 (**)
\$FDOUT[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 (**)
\$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 (**)
\$FDOUT[37]*	It is associated to the yellow LED of TP4i/WiTP Teach Pendant; if lighted it indicates that an application program is running.

(\*\*) In case of TP4i/WiTP Teach Pendant, these \$FDIN/\$FDOUT detect variations only if TPINT Page or Virtual Keyboard are active.

### 5.3.3 \$HDIN

The \$HDIN port array 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.

The array index for the \$HDIN port array corresponds to the DSA board number.

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

The HDIN\_SET and HDIN\_READ built-in routines are used for handing the \$HDIN from a PDL2 program. Also the \$HDIN\_SUSP (field of \$DSA\_DATA) system variable must be considered.

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

### 5.3.4 \$TIMER

The \$TIMER port ARRAY 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.

\$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.

## 5.4 Shared Memory Port Arrays

Shared memory port arrays are used by PDL2 programs to communicate with one another.

A detailed description follows about:

- [\\$BIT](#)
- [\\$WORD](#)

### 5.4.1 \$BIT

The \$BIT port array allows a program access to a bit array, PDL2 treats this as BOOLEAN data. The maximum index value for \$BIT port array is 200.

### 5.4.2 \$WORD

The \$WORD port array 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.

The first 20 elements of \$WORD have predefined meanings of usage and The first 20 elements of \$WORD are mapped to specific system inputs and outputs. Note that these \$WORDS can be used by PDL2 programs as temporary integer storage locations in read and write access, but some of them are continuously updated by the system.

The elements starting at \$WORD[21] are available to the user.

**Fig. 5.1 - \$WORD Mapping (Words 1-20)**

	15	0
1	Input High	
2	Input Low	
3	Output High	
4	Output Low	
5	Input High	
6	Input Low	
7	Output High	
8	Output Low	
9	Input High	
10	Input Low	
11	Output High	
12	Output Low	
13	Input High	
14	Input Low	
15	Output High	
16	Output Low	
17	Input High	
18	Input Low	
19	Output High	
20	Output Low	

The maximum index value for \$WORD port array is 512.

As option it is possible to map \$WORDS to I/O signals related to fieldbus (Profibus, Interbus, DeviceNet), master ad slaves.

## 5.5 System State After Restart

This section describes the state of the C4G system after it has been restarted. There are two different methods of restarting. They are cold start (CCRC) 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, \$GIN, etc. The state of the system is described after each type of shown below restarting.

The \$SDIN, \$SDOUT port arrays are updated to the current state of the system. As far as concernes the other I/O port arrays (\$DIN, \$IN, \$BIT, \$WORD; etc.), the following sections describe which is the state they assume after restart.

### 5.5.0.1 Cold Start

When a cold start is issued from the controller, using CONFIGURE CNTRLER RESTART COLD command (CCRC), the system will come back up as indicated below.

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

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

### 5.5.0.2 Power Failure

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

\$DIN/\$DOUT/\$IN/\$OUT/\$FMI/\$FMO	cleared
\$BIT	retentive bits (1-512) frozen, non-retentive bits cleared
\$AIN/\$AOUT	frozen
\$GIN/\$GOUT	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	frozen

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

## 5.6 User-defined Device Access

User defined port arrays are used for directly accessing to system memory areas. It is important to double check which address is being accessed where using these variables a wrong write operations to some memory locations could damage system memory and require a total software download.

A detailed description follows about:

- [\\$USER\\_BIT](#)
- [\\$USER\\_BYTE](#)
- [\\$USER\\_WORD](#)
- [\\$USER\\_LONG](#)
- [\\$USER\\_ADDR](#)

- \$USER\_LEN
- \$PROG\_UBIT
- \$PROG\_UBYTE
- \$PROG\_UWORD
- \$PROG ULONG
- \$PROG\_UADDR
- \$PROGULEN

### 5.6.1 \$USER\_BIT

The \$USER\_BIT port array allows a program to access bits in memory at an address defined by the user. PDL2 treats this variable as an array of BOOLEAN data.

The starting address from which bits are counted is specified in \$USER\_ADDR [1]. Note that the bits numbering considers the byte in memory to which each bit belongs. For example, \$USER\_BIT[4] is the 4th bit, starting from the right, inside the first byte respect to the address specified in \$USER\_ADDR[1]; \$USER\_BIT[10] is the 2nd bit, starting from the right, of the second byte respect to the address specified in \$USER\_ADDR[1].

### 5.6.2 \$USER\_BYTE

The \$USER\_BYTE port array allows a program to access bytes in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data.

### 5.6.3 \$USER\_WORD

The \$USER\_WORD port array allows a program to access words (2 bytes) in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data.

### 5.6.4 \$USER\_LONG

The \$USER\_LONG port array allows a program to access longwords (4 bytes) in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data.

A precise sequence of operations need to be followed for correctly using these variables. Two other predefined variables must be considered: \$USER\_ADDR and \$USER\_LEN.

Before using \$USER\_BIT, \$USER\_BYTE, \$USER\_WORD, \$USER\_LONG, the corresponding \$USER\_ADDR and \$USER\_LEN element must be properly initialized in order to determine the range of memory that user programs are allowed to access.

### 5.6.5 \$USER\_ADDR

It is an array of four elements where each one defines the starting memory address for:

```
$USER_ADDR[1] : starting memory address for $USER_BIT
$USER_ADDR[2] : starting memory address for $USER_BYTE
$USER_ADDR[3] : starting memory address for $USER_WORD
```

```
$USER_ADDR[4] : starting memory address for $USER_LONG
```

## 5.6.6 \$USER\_LEN

It is an array of four elements where each one defines the length of memory for each:

```
$USER_LEN[1] : length of memory for $USER_BIT
$USER_LEN[2] : length of memory for $USER_BYTE
$USER_LEN[3] : length of memory for $USER_WORD
$USER_LEN[4] : length of memory for $USER_LONG
```

Depending on the user-defined port array, the index defines a different offset respect to the first array element.

For example,

```
$USER_BYTE[5] will access the fifth byte starting from
$USER_ADDR[2];
$USER_LONG[5] will access to the fifth longword starting from
$USER_ADDR[4].
```

If, for example, a portion of 100 bytes of memory need to be accessed starting from the address defined in variable adr\_var, the PDL2 program should be structured like:

```
PROGRAM example NOHOLD
VAR int_var, adr_var : INTEGER
VAR bool_var : BOOLEAN
BEGIN
  -- assign the memory address to adr_var
  adr_var := .....
  -- set the starting address and length of memory for BYTE
  -- memory access
  $USER_LEN[2] := 100
  $USER_ADDR[2] := adr_var
  .....
  -- read the first byte from address adr_var
  int_var := $USER_BYTE[1]
  -- read the fifth byte at address adr_var
  int_var := $USER_BYTE[5]
  -- write a value in the third byte respect to adr_var
  $USER_BYTE[3] := 25
  -- set the starting address and length of memory for BIT
  -- memory access
  $USER_ADDR[1] := adr_var
  $USER_LEN[1] := 100
  -- access to the fifth bit starting from adr_var address
  bool_var := $USER_BIT[5]
END example
```

The previously described user-defined port arrays are available to the entire system and any PDL2 program can use them. This means that multiple programs can contemporaneously access these variables and it is therefore important to avoid interferences in memory access. If for example one program assigns a certain value to \$USER\_ADDR[2] and then another program defines it again with another value, the first program will access to a different memory location when using \$USER\_BYTE.

For solving this kind of problems, another set of user-defined port array variables can be used:

## 5.6.7 \$PROG\_UBIT

The \$PROG\_UBIT port array allows a program to access bits in memory at an address defined by the user. PDL2 treats this variable as an array of BOOLEAN data.

The starting address from which bits are counted is specified in \$PROG\_UADDR[1]. Note that the bits numbering considers the byte in memory to which each bit belongs to. For example, \$PROG\_UBIT[4] is the 4th bit, starting from the right, inside the first byte respect to the address specified in \$PROG\_UADDR[1]; \$PROG\_UBIT[10] is the 2nd bit, starting from the right, of the second byte respect to the address specified in \$PROG\_UADDR[1].

## 5.6.8 \$PROG\_UBYTE

It is used for accessing bytes in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data.

## 5.6.9 \$PROG\_UWORD

It is used for accessing words (2 bytes) in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data.

## 5.6.10 \$PROG ULONG

It is used for accessing longwords (4 bytes) in memory at an address defined by the user. PDL2 treats this variable as an array of INTEGER data. The difference respect to \$USER\_xxx predefined set of variables is that the starting address and the length of memory that can be accessed with \$PROG\_Uxxx variables is local to programs. Each PDL2 program has its own \$PROG\_UADDR and \$PROGULEN and programs cannot interfere each other in the definition of these variables:

## 5.6.11 \$PROG\_UADDR

It is an array of four elements where each one defines the starting memory address for:

\$PROG\_UADDR[1] : starting memory address for \$PROG\_UBIT  
\$PROG\_UADDR[2] : starting memory address for \$PROG\_UBYTE  
\$PROG\_UADDR[3] : starting memory address for \$PROG\_UWORD  
\$PROG\_UADDR[4] : starting memory address for \$PROG ULONG

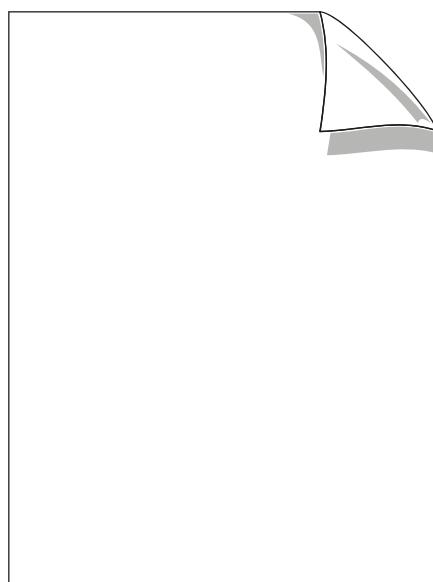
## 5.6.12 \$PROGULEN

It is an array of four elements where each one defines the length of memory for:

\$PROGULEN[1] : length of memory for \$PROG\_UBIT  
\$PROGULEN[2] : length of memory for \$PROG\_UBYTE  
\$PROGULEN[3] : length of memory for \$PROG\_UWORD  
\$PROGULEN[4] : length of memory for \$PROG ULONG

Also for this set of variables it is necessary to first define the starting address and length before using \$PROG\_UBIT, \$PROG\_UBYTE, \$PROG\_UWORD, \$PROG ULONG.

Also refer to [Predefined Variables List](#) chapter.



# 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.

Statements that control execution 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 the UNTIL condition which is always FALSE (so that the cycle is always repeated) and WAIT FORs inside which 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.**

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](#)

Statements that control multiple programs execution are called [Program Synchronization Statements](#).

A detailed description of [Program Scheduling](#) is also available.

## 6.1 Flow Control

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

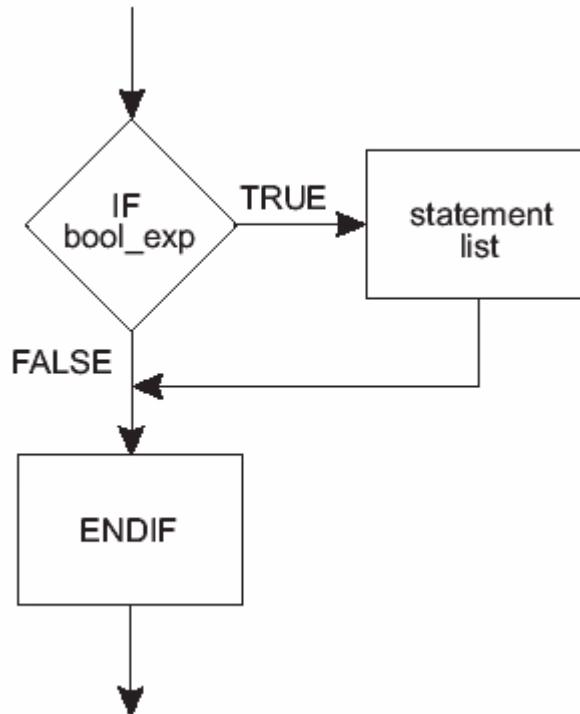
The IF statement allows a program to choose between two possible courses of action, based on the result of a BOOLEAN expression. If the expression is TRUE, the statements following the IF clause are executed. Program control is then transferred to the statement following the ENDIF. If the expression 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 a series of 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.

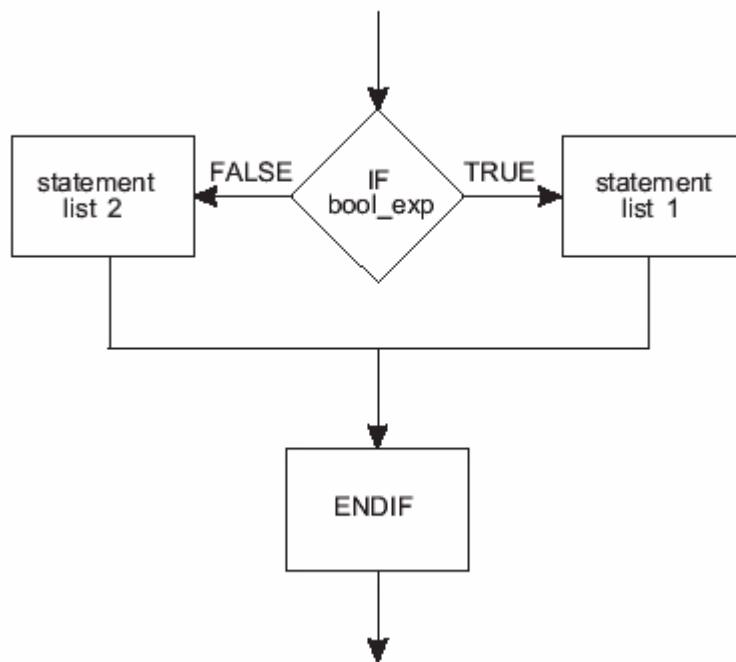
The syntax for an IF statement is as follows:

```
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
    
```

```

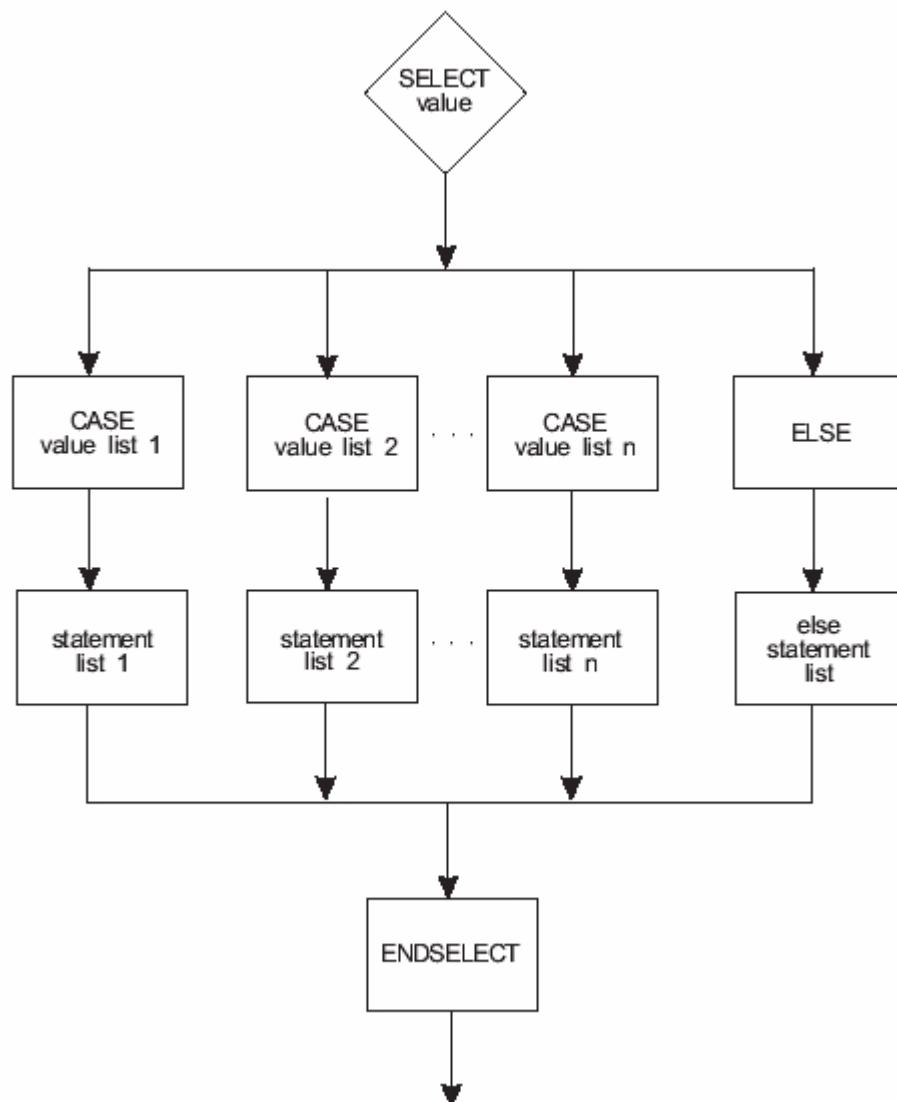
IF part_ok THEN
  good_parts := good_parts + 1
ELSE
  bad_parts := bad_parts + 1
ENDIF
  
```

### 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.

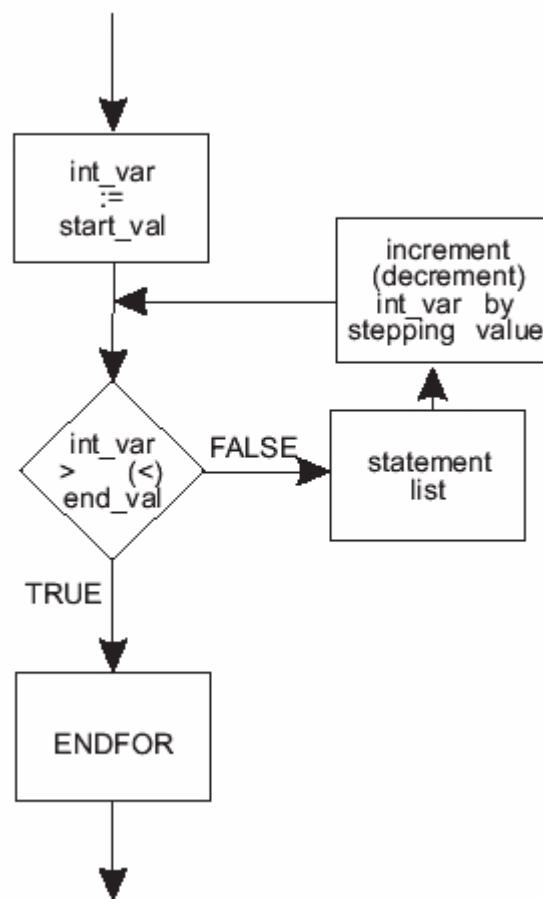
If the loop is written to count down, from the starting value DOWNTO the ending value, a test of greater than or equal is used. If this test initially fails, the loop is skipped. If the STEP value is one, the loop is executed the absolute value of (the ending value - the starting value - 1) times.



**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 | DOWNTO || end_val <STEP step_val>
DO
  <statement>...
ENDFOR
  
```

Example of a FOR statement:

```

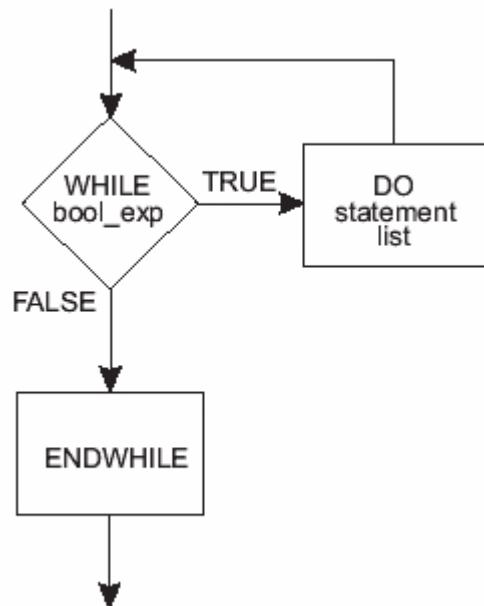
FOR test := 1 TO num_hoses DO
    IF pressure[test] < 170 THEN
        WRITE(Low pres. in hose #, test, Test for leaks., NL)
    ENDIF
ENDFOR
    
```

### 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:

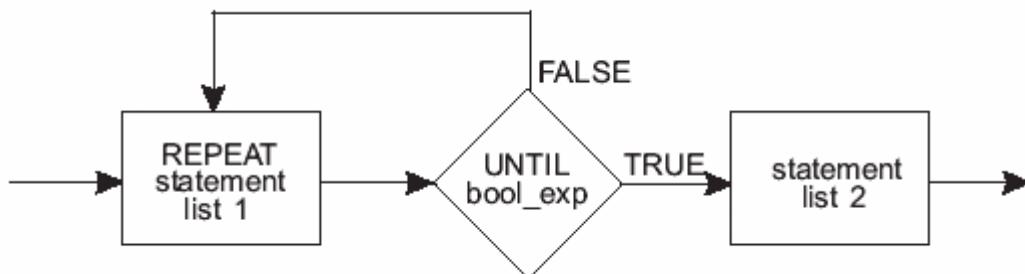
```

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.

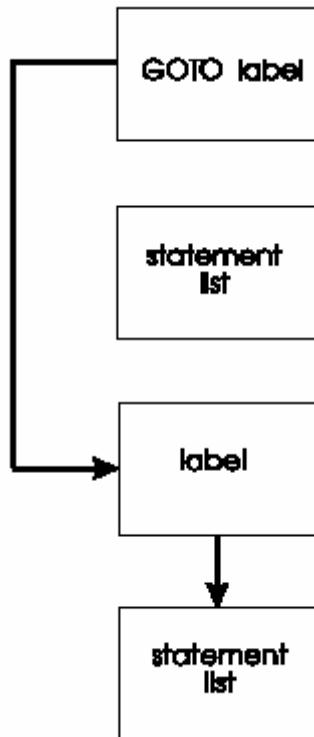
The GOTO statement transfers program control to the place in the program specified by a statement label, a constant identifier at the left margin. The statement label is followed by two consecutive colons (::). Executable statements may follow on the same line, or any line after, the label.

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 ex
    ENDIF
    .
    .
    ex::: WRITE (This is where the GOTO transfers to.)
    .
    .
END prog_1
    
```

## 6.2 Program Control

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.

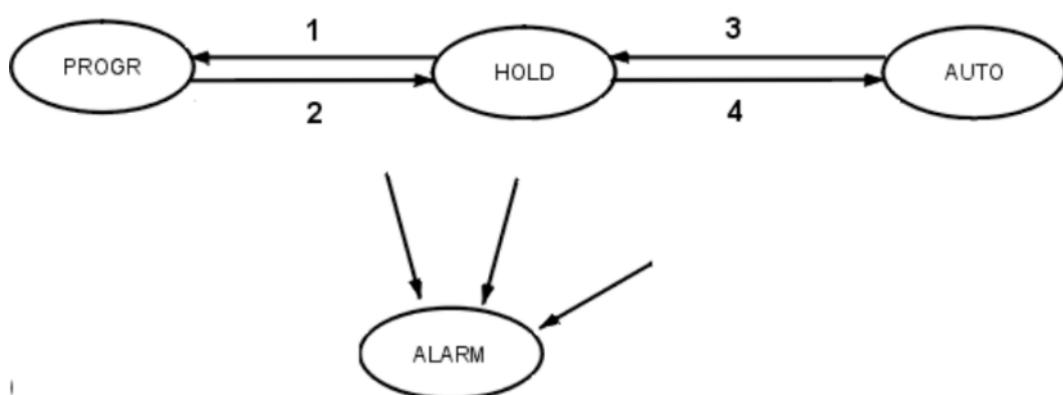
The HOLD or NOHOLD attribute can be specified in the PROGRAM statement to indicate a holdable or non-holdable program. The default program attribute is HOLD.

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 or T2 position + HOLD button released

NOTE that for T2 position Enabling Device must be pressed

PROGR = programming

AUTO = AUTO-T, 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_cntrl
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_ctrl
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 shuld 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

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 serve 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

Since system software version 3.1x, it is allowed to call a function without assigning its relut 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 3-18](#) 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

Since system software version 3.1x, 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)

Since system software version 3.1x, 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

Since the system software version 3.1x, 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.

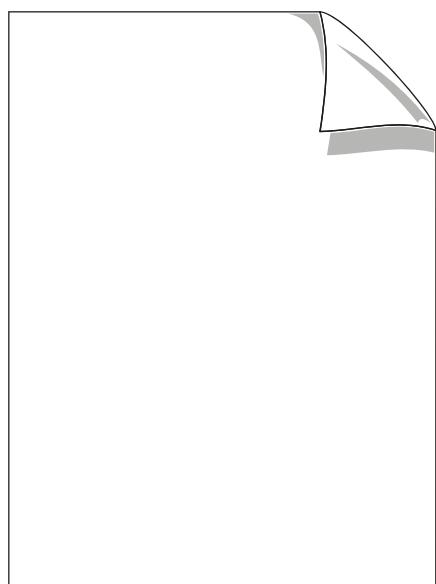
Examples:

```
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, , 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. ly 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 temporarily 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).

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 (ly 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 \$THRDCEXP and \$THRDPARAM 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

### 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, \$SDIN, \$SDOUT, 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
AC_ABORT	Triggered when the ^C key is pressed and the application aid environment is active. Note that the ^C is not felt if the application aid is not active.
AC_CALL_CR T	Triggered when the UTILITY APPLICATION command is issued from the Terminal Window (WinC4G) for activating the application aid environment.
AC_CALL_TP	Triggered when the UTILITY APPLICATION command is issued from the TP for activating the application aid environment.
80	Triggered when WinC4G Program connects to the Control Unit.

EVENT Code	Meaning
81	Triggered when WinC4G Program disconnects from the Controller Unit or an error occurs.
82	Triggered when the EZ softkey on the TP Virtual Keyboard is pressed.
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	Operations Panel key switch turned to T1
102	Operations Panel key switch turned to AUTO
103	Operations Panel key switch turned to REMOTE
106	Transition of the system into PROGR state
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	Triggered when the default arm for jogging is selected issuing the SET ARM TP_MAIN commands
120	ARM button pressed on TP Virtual Keyboard
121	Transition from high to low of HDIN of DSA1
122	Transition from high to low of HDIN of DSA2
123	Transition from high to low of HDIN of DSA3
124	Transition from high to low of HDIN of DSA4
125	Arm 1 is ready to receive offsets from an external sensor
126	Arm 2 is ready to receive offsets from an external sensor

<b>EVENT Code</b>	<b>Meaning</b>
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 WinC4G 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]
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 device is connected on USB port on TP4i/WiTP. I.e. the Disk on Key
156	A device is disconnected on USB port on TP4i/WiTP
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 device is connected on USB port on TP4i/WiTP
178	A device is disconnected from USB port on TP4i/WiTP
179	An I/O port has been forced

EVENT Code	Meaning
180	An I/O port has been unforced
181	An I/O port has been simulated
182	An I/O port has been unsimulated

The use of events related to HDIN (121..124) 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
```

The AC\_CALL\_TP, AC\_CALL\_CRT, AC\_ABORT events are used for the application aid environment, that is a mechanism that can be used for activating and deactivating an application session by issuing the UTILITY APPLICATION command from the system command menu. This implies the existence of an active PDL2 program with enabled conditions containing the following events:

```
WHEN EVENT AC_CALL_TP
WHEN EVENT AC_CALL_CRT DO
WHEN EVENT AC_ABORT DO
```

When one of those events trigger, the associated actions are executed and this is a means offered to program developers to activate their own application programs from the system command menu. When the application environment is active, the system command menu is temporarily disabled until the ^C key is pressed from the device where the UTILITY APPLICATION command was issued. The pressure of the ^C key triggers the AC\_ABORT event, so that the requested actions foreseen when the application environment is exited are undertaken. Note that the AC\_ABORT event only triggers if the application environment has been activated using the described above mechanism and does not trigger a normal ^C key pressure outside this environment (see EVENT 129).

### 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)

WinC4G Program that runs on PC can be used for getting the documentation related to C4G 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 the PROGRAM EDIT or in the MEMORY DEBUG environment;
- pressing the arrow up or arrow down key when working in the EZ 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 the PROGRAM EDIT or MEMORY DEBUG environment when the ^C key is pressed meanwhile the cursor is positioned on the suspendable statement;
- the program that is being executed in the EZ environment when the arrow up/down 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 locally enabled. An error is detected if the ENABLE statement or action includes a condition handler containing other motion events. Locally 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 locally 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\_ERROR predefined variable. It is initialized to 0 unless the interrupt routine is activated as an action of an error event. 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](#).

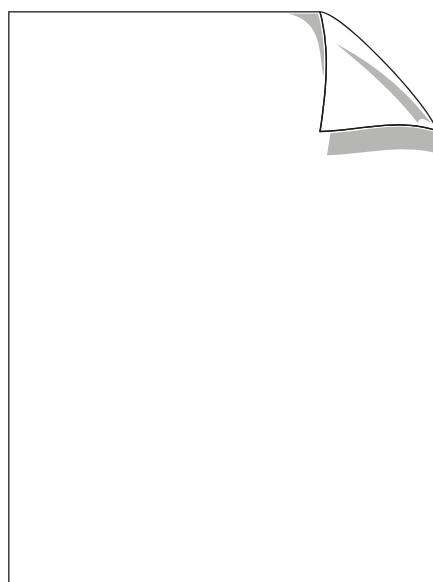
#### 8.4.14 Execution Order

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. SERIAL INPUT/OUTPUT

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 WinC4G, is provided by COMAU to be run on a PC (for further information see Chapter **WinC4G Program** in the **Use of C4G Control Unit** 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.

## 9.1 Serial Devices

PDL2 supports the following types of serial devices:

- [WINDOW Devices](#)
- [FILE Devices](#)
- [PIPE Devices](#)
- external;
- [COMMUNICATION 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 TP4i/WiTP (scrolling area of TP-INT Page) and WinC4G (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(WinC4G Terminal Window )	Teach Pendant ( scroll area in TP-INT Page)
CRT: (default)	TP: (default)
CRT1:	TP0:
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 WinC4G and TP4i/WiTP 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: (TP4i/WiTP 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 disk on key is inserted in the USB port of TP4i/WiTP Teach Pendant

- XD: is an eXternal device which is recognized by the system when the disk on key is inserted in the USB port.

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 DV4\_CNTRL(1) and can be deleted with DV4\_CNTRL (2). For further details see [par. 11.40 DV4\\_CNTRL Built-In Procedure on page 11-27](#).

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 upto the user. What is written - is read.

### 9.1.4 COMMUNICATION Devices

Communication devices are ports through which programs can communicate with other devices connected to the Controller.

PDL2 recognizes the following communication device names:

COM1:	COM2:	COM3:	COMP:
DGM1:	DGM2:	DGM3:	USBC:
NET1:	NET2:	NETP:	NETT:
MDM:			NETU:
PPP1:	PPP2:	PPP3:	PPPM:

COMP: is the channel for WinC4G Program interfacing.

COM1:..COM3: are serial ports that are available for the user.

NETn: are the channels of communication for various local area networking configurations:

NET0: is the default device upon which WinC4G 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

PPP1..PPP3: are the PPP mounted devices

PPPM: is the device when PPP is mounted on the Modem device.

Communication 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.

[DV4\\_CNTRL Built-In Procedure](#) can be used for getting and setting the communication 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

### 9.1.5 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.

### 9.1.6 ATTACH and DETACH Statements

The ATTACH statement allows a program to gain exclusive control of a device so that other programs cannot access that device. The DETACH statement releases exclusive control of a device that has been attached.

The syntax for the ATTACH and DETACH statements when applied to I/O devices are as follows:

```
ATTACH device_str <, device_str>
DETACH device_str <, device_str>
```

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:
DGM0 :	TP3:

Communication Devices:	Window Devices:
DGM1:	
DGM2:	
DGM3:	
DGM4:	
For example:	
	ATTACH 'COM1:', 'DGM4:'
	DETACH 'TP2:'
	DETACH 'abcd:' -- release the user-defined window
	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.

## 9.2 Logical Unit Numbers

A logical unit number (LUN) represents a connection between a program and a physical device with which the program can communicate.

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 WinC4G 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.

### 9.2.1 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 file stats.dat,
read only
OPEN FILE comm_lun ('COM1:', 'R') -- opens comm port COM1:, read
only
OPEN FILE win_lun ('abcd:', 'RW') -- opens user-defined window
ABCD:
```

## 9.2.2 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_ADLM	\$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 ENDOOPEN 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_ADLMNT = '\010\013\011\027\003',
ENDOPEN
```

### 9.2.3 CLOSE FILE Statement

The CLOSE FILE statement closes a LUN, ending the connection between the program and the device. Any buffered data is written to the device before the CLOSE FILE statement is executed.

The syntax of the CLOSE FILE statement is as follows:

```
CLOSE FILE || lun_var | ALL ||
```

The *lun\_var* can be any user-defined integer variable that represents an open LUN. Specifying ALL instead of a lun will close all files opened by the program.

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.

### 9.2.4 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
```

#### 9.2.4.1 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.

#### 9.2.4.2 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.

### 9.2.5 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.

### 9.2.5.1 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.

### 9.2.5.2 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.

### 9.2.5.3 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.

# 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
- 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 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.

*device\_str* can be any of the following system device name strings or a user-defined window device:

Communication Devices	Window Devices
COM1:	CRT1:
COM2:	CRT2:
COMP:	CRT3:
	TP0:
	TP1:
	TP2:

<b>Communication Devices</b>	<b>Window Devices</b>
------------------------------	-----------------------

---

 TP3:

When an I/O device is attached, other programs cannot open a LUN on that device. 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
ROUTINE r2 EXPORTED FROM p2
ROUTINE r1
BEGIN
  r2
END r1
BEGIN
  r1
END p1

PROGRAM p2
ROUTINE r2 EXPORTED FROM p2
ROUTINE r2
BEGIN
  -- ATTACH statement
END r2
BEGIN
END p2

```

*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:*

```
PROGRAM example_1
-- declaration section
BEGIN
-- executable section
END example1
PROGRAM example3
-- declaration section
BEGIN
-- initial executable
section
CYCLE
-- cycled executable
section
END example3
```

```
PROGRAM example2
-- declaration section
BEGIN CYCLE
-- executable section
END example2
```

*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 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 7-8.](#)

## 10.7 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](#)

## 10.8 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. CLOSE FILE ALL causes all files currently opened by the program to be closed. Files 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](#)  
[Serial INPUT/OUTPUT Chapter](#)

## 10.9 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](#)

## 10.10 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](#)

## 10.11 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](#)

## 10.12 CYCLE Statement

The CYCLE statement allows the programmer to create a continuous cycle.

*Syntax:*

CYCLE

*Comments:*

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
  -- cycled executable
  --   section
END example1
```

```
PROGRAM example2
  -- declaration section
BEGIN CYCLE
  -- executable section
END example2
```

*See also:*

[BEGIN Statement](#)  
[EXIT CYCLE Statement](#)

## 10.13 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](#)

## 10.14 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 (refer to [Serial INPUT/OUTPUT](#) chapter).

*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](#)

[Serial INPUT/OUTPUT Chapter](#)

## 10.15 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](#)

## 10.16 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.

*device\_str* can be any of the following system devices or a user-defined window device:

Communication Devices	Window Devices
COM1:	CRT1:
COM2:	CRT2:
COMP:	CRT3:
	TP0:
	TP1:
	TP2:
	TP3:

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 \(ATTACH Attribute\)](#)

## 10.17 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](#)

## 10.18 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; assignment, 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 compromissing 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](#).

## 10.19 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](#)

## 10.20 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 (refer to [Serial INPUT/OUTPUT](#) chapter). 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 then 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](#)  
[Serial INPUT/OUTPUT Chapter](#)

## 10.21 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](#)

## 10.22 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](#)

## 10.23 FOR Statement

The FOR statement executes a sequence of statements a known number of times.

*Syntax:*

```
FOR intvar := startval || TO | DOWNT0 || 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 DOWNT0 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](#)

## 10.24 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.

GOTO statements should be used with caution. In most cases where execution flow control is needed in a program, it can be done with the other flow control statements (FOR, WHILE, REPEAT, IF, etc.) in the language.

*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](#)

## 10.25 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 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](#)

## 10.26 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](#)

## 10.27 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 3-19](#)); 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 3-18](#)

## 10.28 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](#)

## 10.29 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](#)

## 10.30 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](#)

## 10.31 OPEN FILE Statement

The OPEN FILE statement opens a LUN on a specified device, establishing a connection between the program and the device through which I/O operations can be performed.

*Syntax:*

```
OPEN FILE lun_var (device_str, access_str) <with_clause>
```

*Comments:*

lun\_var can be any user-defined INTEGER variable. It is assigned a logical unit number (LUN) by the system that can be used in subsequent I/O operations to specify the device.

*device\_str* can be any STRING expression representing a device. File devices also include a file name and extension. The default file device is 'UD:'.

*access\_str* can be any STRING expression representing the access with which the device is to be opened. Valid access designations are as follows:

R	-- read only
W	-- write only
RW	-- read and write
WA	-- write append
RWA	-- read and write append



**Note that in case of RW, the file must already exist, otherwise an error is returned.**

A LUN remains opened until it is closed with a CLOSE FILE statement, program execution is completed, or the program is deactivated.

The optional WITH clause can designate temporary values for predefined variables related to LUNs (listed in [Chap.9. - Serial INPUT/OUTPUT](#)).

`WITH predef_lun_var = value <, predef_lun_var = value>...`

If an OPEN FILE statement needs more than a single line, commas must be used to end a line after the closing parenthesis or after each WITH clause. The reserved word ENDOPEN must then be used to indicate the end of the statement. Refer to the examples below.

*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
```

*See also:*

[CLOSE FILE Statement](#)

[READ Statement](#)

[WRITE Statement](#)

[Chap.9. - Serial INPUT/OUTPUT](#)

## 10.32 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](#)

## 10.33 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 unpause.

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_cntrl
```

```
PAUSE ALL
```

*See also:*

[UNPAUSE Statement](#)

## 10.34 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 and DETACH for non-holdable programs. Note that the DETACH attribute cannot be used on EZ 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.

**EZ**
**Syntax:**
**EZ**

The use of this attribute on the program header causes several changes in the way the program is handled by the system. First, a program with this attribute is always loaded with the /FULL option. Second, multiple statements are not allowed on the same line. Third, a routine is implicitly defined when the call statement is inserted. Fourth, the EZ attribute is required when the EZ environment is used. Finally, the EXPORTED FROM clause is automatically added to the routine declaration.

**Examples:**

```

PROGRAM example1 PROG_ARM = 2, HOLD
    -- VAR and CONST section
BEGIN
    .
    .
    .
END example1

PROGRAM example2 PRIORITY = 3, NOHOLD

PROGRAM example3 PLC

```

**See also:**

[BEGIN Statement](#)  
[END Statement](#)

## 10.35 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. - INPUT/OUTPUT Port Arrays](#)

## 10.36 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](#)

## 10.37 READ Statement

The READ statement reads input data into program variables from the specified LUN.

*Syntax:*

```
READ <lun_var> (var_id <,var_id>...)
```

*Comments:*

*lun\_var* can be a variable representing any open LUN or either of the predefined

LUNs, LUN\_CRT, LUN\_TP or LUN\_NULL.

If a *lun\_var* is not specified, the default LUN indicated by \$DFT\_LUN is used.

*var\_id* can be any variable identifier of the following data types:

INTEGER	VECTOR
REAL	POSITION
BOOLEAN	JOINTPOS
STRING	XTNDPOS

*var\_id* can also be a predefined variable reference as long as the predefined variable does not have value limits. Refer to [Chap.12. - Predefined Variables List](#) to determine if a predefined variable has limits.

The reserved word NL can also be used for *var\_id*. When NL is used, the remainder of the current input line is ignored.

Each data item that is read is assigned to the corresponding variable from the list of identifiers. The data item that is read must be of the same type as the corresponding identifier passed to the READ. An incompatible type will cause an error to be returned.

Optional [Format Specifiers](#) can be used with each *var\_id* to format input as explained in [Chap.9. - Serial INPUT/OUTPUT](#).

READ is an asynchronous statement which means it continues while the program is paused and interrupt service routines can begin before the statement completes. The predefined variable \$READ\_TOUT indicates the timeout period for asynchronous reads.

When reading from a communication port, if the READ delimiter consists of multiple characters, the DV4\_CNTRL built-in procedure should be called for assigning the COM\_RDAHD attribute to the port.

*Examples:*

```

READ (x, y, z)          -- lun_var defaults to $DFT_LUN

READ LUN_TP (length, height, weight)

OPEN FILE crt1_lun ('CRT1:', 'RW')
READ crt1_lun (menu_selection)

DV4_CNTRL ('COM1:', COM_RDAHD)
-- Set ASCII delimiter to carriage return followed by line
-- feed
OPEN FILE com1_lun('COM1:', 'r') WITH $FL_ADLM = '\013\010'
READ com1_lun (str::8)

```

*See also:*

[DV4\\_CNTRL Built-In Procedure](#)  
[DECODE Statement](#)  
[OPEN FILE Statement](#)  
[WRITE Statement](#)  
[Chap.9. - Serial INPUT/OUTPUT](#)

## 10.38 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](#)

## 10.39 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](#)

## 10.40 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](#)

## 10.41 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](#)

## 10.42 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](#)

## 10.43 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](#)

## 10.44 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](#)

## 10.45 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](#)

## 10.46 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](#)

## 10.47 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](#)

## 10.48 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>
```

*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 *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](#)

## 10.49 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

# 10.50 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](#)

# 10.51 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](#)

## 10.52 WRITE Statement

WRITE statement writes output data from a program to the specified LUN.

Syntax:

WRITE <lun\_var> (<expr> <, <expr>>...)

Comments:

*lun\_var* can be a variable representing any open LUN or either of the predefined LUNs: LUN\_CRT, LUN\_TP, or LUN\_NULL.

If *lun\_var* is not specified, the default output LUN, indicated by \$DFT\_LUN, is used. *expr* can be any expression of the following data types:

INTEGER	VECTOR
REAL	POSITION
BOOLEAN	JOINTPOS
STRING	XTNDPOS

The reserved word NL can also be used. When specified, a carriage return is written so the following items begin on the next line of output.

Each *expr* is written out in the order that it is listed.

Optional [Format Specifiers](#) can be used with each *expr* to format output as explained in [Chap.9. - Serial INPUT/OUTPUT](#).

Writing to certain types of devices (i.e. communication port) is done asynchronously so other programs can execute simultaneously. The predefined variable \$WRITE\_TOUT specifies the time out period for asynchronous writes.

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:

[OPEN FILE Statement](#)  
[READ Statement](#)  
[Chap.9. - Serial INPUT/OUTPUT](#)

# 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_MOVE_ATVEL Built-in Procedure ARM_NUM Built-In Function 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_SET_OFST 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 DV4_CNTRL Built-In Procedure DV4_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_SET_INT Built-In Procedure STR_SET_REAL Built-In Procedure 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 v3.11 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 IS_FLY Built-In Function PROG_OWNER Built-In Function PROG_STATE Built-In Function RPLC_GET_IDX Built-In Procedure 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 TABLE_ADD Built-In Procedure TABLE_DEL Built-In Procedure VAR_INFO Built-In Function VAR_UNINIT Built-In Function

## 11.1 ABS Built-In Function

The ABS built-in function returns the absolute value of a specified number.

<b>Calling Sequence:</b>	ABS (number)
<b>Return Type:</b>	REAL   INTEGER
<b>Parameters:</b>	number :    REAL   INTEGER    [IN]
<b>Comments:</b>	<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>
<b>Examples:</b>	<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.

<b>Calling Sequence:</b>	ACOS (number)
<b>Return Type:</b>	REAL
<b>Parameters:</b>	number :REAL [IN]
<b>Comments:</b>	<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>
<b>Examples:</b>	<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.

<b>Calling Sequence:</b>	ACT_POST (erronum_int, error_string)
<b>Parameters:</b>	erronum_int :INTEGER [IN] error_string :STRING [IN]
<b>Comments:</b>	<i>erronum_int</i> is the number of the error to be posted <i>error_string</i> is the message to be posted, associated to <i>erronumber_int</i>
<b>Examples:</b>	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>)`

<b>Return Type:</b>	INTEGER
<b>Parameters:</b>	index : INTEGER [ IN] by_reference: BOOLEAN [ IN] size_array1 : INTEGER [ IN] size_array2 : INTEGER [ IN]
<b>Comments:</b>	<i>index</i> specifies the argument index in the parameters list. <i>by_reference</i> indicates whether the argument is passed by reference or not. <i>size_array1</i> specifies the array 1st dimension (if the argument is an array). <i>size_array2</i> 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.

<b>Calling Sequence:</b>	ARG_SET_VALUE (index, variable <, array_index1> <, array_index2>)
<b>Parameters:</b>	index : INTEGER [ IN] variable: ANY TYPE [ IN] array_index1 : INTEGER [ IN] array_index2 : INTEGER [ IN]
<b>Comments:</b>	<i>index</i> specifies the argument index in the optional list . <i>variable</i> indicates the variable containing the value to be set to the specified argument. <i>array_index1</i> specifies the array row (if the argument is an array) <i>array_index2</i> specifies the array column (if the argument is a bidimensional array)
<b>Examples:</b>	see examples in <a href="#">ARG_GET_VALUE Built-in Procedure</a> section.
<b>See also:</b>	<a href="#">ARG_COUNT Built-In Function</a> <a href="#">ARG_INFO Built-In Function</a> <a href="#">ARG_GET_VALUE Built-in Procedure</a>

## 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.

<b>Calling Sequence:</b>	ARM_COLL_THRS (arm_num, coll_type, time<, safety_gap>)
<b>Parameters:</b>	arm_num : INTEGER [ IN] coll_type : INTEGER [ IN] time : INTEGER [ IN] safety_gap: BOOLEAN [ IN]
<b>Comments:</b>	<i>arm_num</i> is the arm on which the acquisition should be applied to. <i>coll_type</i> is the type of sensitivity to be acquired. This routine directly reads \$ARM_SENSITIVITY variable in relation to the specified <i>coll_type</i> . The allowed values are COLL_LOW to COLL_USER10. <i>time</i> is the duration in seconds of the acquisition and should correspond, at least, to the direction 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, starting from system software 2.42 and subsequent versions, 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:**

<code>flag</code>	: BOOLEAN	[ IN]
<code>positioner_arm</code>	: INTEGER	[ IN]
<code>working_arm</code>	: 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.

<b>Calling Sequence:</b>	ARM_GET_NODE <(arm_num)>
<b>Return Type:</b>	INTEGER
<b>Parameters:</b>	arm_num : INTEGER [ IN ]
<b>Comments:</b>	<p>If <i>arm_num</i> is not specified, the default program arm is used.</p> <p>The value returned indicates the next path node to be processed for the arm.</p> <p>Path processing is ahead of the motion so this does not necessarily indicate the next path node motion.</p> <p>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.</p>
<b>Examples:</b>	<pre>ROUTINE skip_node(arm_num, skip_num: INTEGER) VAR node_num : INTEGER BEGIN     node_num := ARM_GET_NODE(arm_num)     WHILE( node_num &lt;&gt; skip_num) AND node_num &gt; 0 ) DO         node_num := ARM_GET_NODE(arm_num)     ENDWHILE     IF node_num &gt; 0 THEN         ARM_SET_NODE( skip_num + 1, arm_num)     ELSE         WRITE( 'ERROR: Can not skip the node', NL)     ENDIF END skip_node</pre>

## 11.11 ARM\_JNTP Built-In Function

The ARM\_JNTP built-in function returns the current JOINTPOS value for a specified arm.

<b>Calling Sequence:</b>	ARM_JNTP <(arm_num)>
<b>Return Type:</b>	JOINTPOS
<b>Parameters:</b>	arm_num : INTEGER [ IN ]
<b>Comments:</b>	If <i>arm_num</i> is not specified, the default arm is used.
<b>Examples:</b>	<pre>PROGRAM reset VAR     zero, robot : JOINTPOS     i : INTEGER BEGIN     robot := ARM_JNTP     FOR i := 1 TO 6 DO         zero[i] := -robot[i]     ENDFOR</pre>

```

MOVE TO zero
END reset

```

## 11.12 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.

Available software option: **C4G Speed Control for Arm Motion** (for the options codes, please refer to the **C4G Control Unit Technical Specifications** Manual, chapter **Software Options**).

**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:**

*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).

## 11.13 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

```

## 11.14 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, \$UFRAAME.  
If *arm\_num* is not specified, the default 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

```

## 11.15 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

```

## 11.16 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 **C4G 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:**

flag : BOOLEAN	[ IN ]
ax1, ax2, ax3, ax4, ax5, ax6: INTEGER	[ IN ]
arm_num : 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.

## 11.17 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
```

## 11.18 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
```

## 11.19 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
  END print_2dim_ary

```

## 11.20 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 degrees.  
*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

```

## 11.21 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 ]

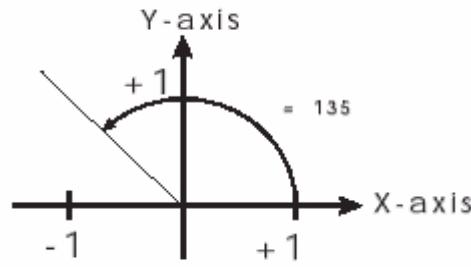
**Comments:** The arc tangent is measured in degrees.  
The result is in the range of -180 to 180.  
The quadrant of the point (x, y) determines the sign of the result.  
If x and y are both zero an error occurs.

**Examples:**

```

x := ATAN2(0, 0)  -- ERROR
x := ATAN2(1, -1) -- x = 135 (see diagram)

```



## 11.22 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>)

<b>Parameters:</b>	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

```

## 11.23 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 enabled (Safety Interlock Module).

**Note that** the program which switches to DRIVE ON must be **the same** which switches to DRIVE OFF.



The **AUX\_DRIVES** built-in Procedure can be used only if the Sik Device associated to the involved axes/arms has been properly configured. For further details refer to the Use of C4G Controller Unit manual (IO\_TOOL Program, IO\_INST Program, Setup Page chapters), Integration guidelines Safeties, I/O, Communications manual, and par. 5.3.1 \$SDIN and \$SDOUT on page 5-6 in current manual.

**Calling Sequence:** `AUX_DRIVES (enable_flag, axis_num <,arm_num>)`

**Parameters:**

<code>enable_flag</code>	: BOOLEAN	[ IN ]
<code>axis_num</code>	: INTEGER	[ IN ]
<code>arm_num</code>	: 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 positioner.



If '0' it means ALL robot axes which are NOT specified as AUXILIARY axes.

**Examples:**

```

AUX_DRIVES (ON, 7, 1) -- enables the first positioner, linked to axes 7 and 8
AUX_DRIVES (OFF, 9, 1) -- disables the second positioner, linked to axes 9 and 10
MOVE TO pnt0001j -- the robot (axes 1..6) and the first
positioner (axes 7..8) are moved
  
```

## 11.24 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:**

<code>flag</code>	: BOOLEAN	[ IN ]
<code>aux_axis</code>	: INTEGER	[ IN ]
<code>arm_num</code>	: 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)
  
```

## 11.25 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:**

var_ref:	INTEGER	[ IN, OUT ]
bit_num:	INTEGER	[ IN ]
bool_test:	BOOLEAN	[ IN ]
op_set_clear:	BOOLEAN	[ IN ]
bool_value:	BOOLEAN	[ IN ]

**Comments:**

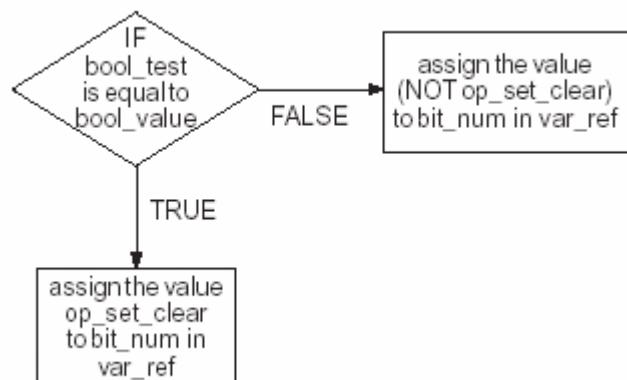
*var\_ref* is an INTEGER variable or port reference. A bit of this variable will be set by this built-in.

*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)
```

## **11.26 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*)

## 11.27 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, \$GIN, \$GOUT, \$WORD, \$USER\_BYTE, \$USER\_WORD, \$USER\_LONG, \$PROG\_UBYTE, \$PROG\_UWORD, \$PROG ULONG.

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.

**Examples:**

```
CONDITION[1]:  
    WHEN BIT_FLIP ($WORD[100], 4, ON) DO  
        HOLD  
    ENDCONDITION  
    ENABLE CONDITION[1]  
    . . . . .  
    WAIT FOR ($USER_BYTE[3], 5, FALSE)
```

## 11.28 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)

## 11.29 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)

## 11.30 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
```

## 11.31 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:**

```
cur_time := CLOCK
WRITE(CLOCK, NL) -- say time is 1514682000
-- 30 second time interval
WRITE(CLOCK, NL) -- now time is 1514682030
```

## 11.32 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.

<b>Calling Sequence:</b>	COND_ENABLED (cond_num <,prog_name >)
<b>Return Type:</b>	BOOLEAN
<b>Parameters:</b>	cond_num : INTEGER [ IN] prog_name: STRING [ IN]
<b>Comments:</b>	<i>cond_num</i> is the number of the condition to be tested. <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. The returned value will be TRUE if the condition <i>cond_num</i> is enabled (locally or ly) and FALSE if the condition is disabled. An error occurs if the condition is not defined
<b>Examples:</b>	<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>

## 11.33 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.

<b>Calling Sequence:</b>	COND_ENBL_ALL (cond_map <,prog_name >)
<b>Parameters:</b>	cond_map : ARRAY OF INTEGER [ IN, OUT] prog_name: STRING [ IN]
<b>Comments:</b>	<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 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. <i>prog_name</i> 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 0x3FFFFFFF <> 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

## 11.34 CONV\_SET\_OFST Built-In Procedure

The CONV\_SET\_OFST built-in procedure sets the shift distance of the conveyor base.

**Calling Sequence:** CONV\_SET\_OFST (distance, conv\_tbl\_idx <,arm\_num >)

**Parameters:**

distance:	REAL	[ IN ]
conv_tbl_idx:	INTEGER	[ IN ]
arm_num:	INTEGER	[ IN ]

**Comments:** *distance* is the distance between the workpiece and the sensor which detected its passage on the conveyor. In case of circular conveyor, the distance must be calculated along the circumference. *conv\_tbl\_idx* is the index of the conveyor table. *arm\_num* is the arm number. If not specified, the default arm is used. Refer to the "Motion Control" chapter in the "Motion Programming" manual for further details.

**Examples:**

```
-- set the shift for conveyor 1 of arm 2 to 1000.5
CONV_SET_OFST (1000.5, 1, 2)
```

## 11.35 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)

## 11.36 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 := 0b000111001111100010110011001010
  .
  .
  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
```

## 11.37 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

## 11.38 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:\appl\arc, the following statements:

```

DIR_SET('UD:')
SYS_CALL('FUDM','prova')
will create UD:\app\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.**

## 11.39 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

## 11.40 DV4\_CNTRL Built-In Procedure

The DV4\_CNTRL built-in procedure is used to control a communication device.

**Calling Sequence:** DV4\_CNTRL (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
- 14..17   reserved

- 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 DV4\_CTRL 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.

- An INTEGER for the linger time
  - An INTEGER for the size of the “send buffer”
  - An INTEGER for the size of the “receive buffer”
- 29 Accept on a port for a connection.  
 Parameters:  
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest
- 30 Establish a TCP/IP connection.  
 Parameters:  
 - An INTEGER which is the LUN (see [OPEN FILE Statement](#)) associated to the channel of interest  
 - A STRING name of the remote host  
 - An INTEGER for the remote port number (0xFFFFFFFF).
- 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

### Examples

Use of code 28:

if the user would like both to enable the NO\_DELAY functionality and disable the NOT\_KEEPALIVE functionality, the DV4\_CTRL built-in routine is to be called as follows:

```
DV4_CTRL ( 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.

## 11.41 DV4\_STATE Built-In Function

This function returns information about a specified device.

**Calling Sequence:** DV4\_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:	Winc4g 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	Winc4g on serial connection
	9	3964r protocol
	10	FTP Network protocol
	11	Winc4g on Ethernet connection
16-32		reserved

**Examples:** inform\_int := DV4\_STATE(dev\_name\_str)

## 11.42 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.

<b>Calling Sequence:</b>	EOF ( <i>lun_id</i> )
<b>Return Type:</b>	BOOLEAN
<b>Parameters:</b>	<i>lun_id</i> : INTEGER [ IN ]
<b>Comments:</b>	<p><i>lun_id</i> can be any user-defined variable that represents an open logical unit number (LUN).</p> <p>TRUE will be returned if the last operation on <i>lun_id</i> was a READ and the end of file was reached before all specified variables were read.</p> <p>If the end of file is reached before the READ statement completed, all variables not yet read are set to uninitialized.</p> <p>If <i>lun_id</i> 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.</p>
<b>Examples:</b>	<pre> OPEN FILE <i>lun_id</i> ('data.txt', 'R') READ <i>lun_id</i> (<i>str</i>) WHILE NOT EOF(<i>lun_id</i>)     WRITE (<i>str</i>, NL)     READ <i>lun_id</i> (<i>str</i>) ENDWHILE  OPEN FILE <i>lun_id2</i> ('COM0:', r) WITH \$FL_NUM_CHARS = 1 bool_var := FALSE WHILE NOT bool_var DO     READ <i>lun_id2</i> (<i>str</i>::1)     IF <i>str</i> = '\033' THEN -- assume '!' as end of file         WRITE ('Found EOF', NL)         CLOSE FILE <i>lun_id2</i>     ENDIF ENDWHILE </pre>

## 11.43 ERR\_POST Built-In Procedure

The ERR\_POST built-in procedure causes a user-defined error. It handles active alarms: these are alarms which require responses by the user. When the user has selected the response, the program is notified (signal event or variable set).

<b>Calling Sequence:</b>	ERR_POST ( <i>error_num</i> , <i>error_str</i> , <i>error_sev</i> <, <i>post_flags</i> > <, <i>resp_flags</i> > <, <i>alarm_id</i> > <, <i>resp_data</i> > <, <i>parameters</i> >)														
<b>Parameters:</b>	<table border="0"> <tr> <td><i>error_num</i> : INTEGER</td> <td>[ IN ]</td> </tr> <tr> <td><i>error_str</i> : STRING</td> <td>[ IN ]</td> </tr> <tr> <td><i>error_sev</i> : INTEGER</td> <td>[ IN ]</td> </tr> <tr> <td><i>post_flags</i> : INTEGER</td> <td>[ IN ]</td> </tr> <tr> <td><i>resp_flags</i> : ARRAY OF INTEGERS</td> <td>[ IN ]</td> </tr> <tr> <td><i>alarm_id</i> : INTEGER</td> <td>[ OUT ]</td> </tr> <tr> <td><i>resp_data</i> : INTEGER</td> <td>[ OUT ]</td> </tr> </table>	<i>error_num</i> : INTEGER	[ IN ]	<i>error_str</i> : STRING	[ IN ]	<i>error_sev</i> : INTEGER	[ IN ]	<i>post_flags</i> : INTEGER	[ IN ]	<i>resp_flags</i> : ARRAY OF INTEGERS	[ IN ]	<i>alarm_id</i> : INTEGER	[ OUT ]	<i>resp_data</i> : INTEGER	[ OUT ]
<i>error_num</i> : INTEGER	[ IN ]														
<i>error_str</i> : STRING	[ IN ]														
<i>error_sev</i> : INTEGER	[ IN ]														
<i>post_flags</i> : INTEGER	[ IN ]														
<i>resp_flags</i> : ARRAY OF INTEGERS	[ IN ]														
<i>alarm_id</i> : INTEGER	[ OUT ]														
<i>resp_data</i> : 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 12. These values represent the following error severities:

- 2 : Warning
- 4 : Pause, hold if holdable
- 6 : Pause all, hold if holdable
- 8 : Hold
- 10 : DRIVE OFF, pause all, deactivate
- 12 : DRIVE OFF, deactivate all

When ERR\_POST is executed, the error is treated like any system error. The error is posted to the error log and any error-specific condition handlers are scanned. The \$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 post in the ERROR.LOG file
- Bit 2: Do not post on the status window of the system screen
- Bit 3: Do not post to condition handler scanning process
- Bit 4: Do not post on the scrolling window of the TP system screen
- Bit 5: Do not post on the scrolling window of the PC screen
- Bit 6: Do not post to the system state machine
- Bit 7: Cause program state transition based on error severity (if bit 7 isn't set the transition won't take place)
- Bit 10: Automatic return from current routine (Bit 7 must also be set)
- Bit 12: Do not post to error logger task
- Bit 13: Add to binary log even if severity 2
- Bit 15: A latched ALARM
- Bit 16: A latched ALARM with disabled DRIVE ON
- Bit 17: A latched ALARM with disabled AUTO state transition

*resp\_flags* : An array of integers with a min. size of 4 elements:

- [1] : A mask with the available responses that the user can select (see constant ERR\_OK etc)
- [2] : Default response
- [3] : Action to be performed when the user has acknowledged the alarm eg. Signal event or set a variable
  - 0 = Cancel the active alarm
  - 1 = Post an event with data the event to be raised
  - 2 = Set a variable. When the active alarm is acknowledged the variable is with the low word the alarm id, high word the response
- [4] : The data associated with the action eg. variable to be set or event id to be raised

*alarm\_id* : Optional variable (by reference) that will contain the alarm identifier after the alarm has become active

*resp\_data* : optional variable (by reference) that will contain the response made by the user with low word the *alarm\_id* and high word the response.

**Examples:**

```

PROGRAM actalarm NOHOLD

VAR vi_id, vi_data : INTEGER
    wi_conf : ARRAY[4] OF INTEGER

BEGIN
CYCLE
    --First of all configure how the alarm is to be displayed
    --Options to be displayed on menu
    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 8::8::2, NL)
    --Now cancel the alarm
    CANCEL ALARM vi_id
END actalarm

```

## 11.44 ERR\_STR Built-In Function v3.11

This built-in function returns the message string associated to a specific error.

<b>Calling Sequence:</b>	str := ERR_STR(err_num<,err_sev,err_flags> <,language_int>)												
<b>Return Type:</b>	STRING												
<b>Parameters:</b>	<table border="0"> <tr> <td>err_num:</td> <td>INTEGER</td> <td>[ IN ]</td> </tr> <tr> <td>err_sev:</td> <td>INTEGER</td> <td>[ OUT ]</td> </tr> <tr> <td>err_flags:</td> <td>INTEGER</td> <td>[ OUT ]</td> </tr> <tr> <td>language_int:</td> <td>INTEGER</td> <td>[ IN ]</td> </tr> </table>	err_num:	INTEGER	[ IN ]	err_sev:	INTEGER	[ OUT ]	err_flags:	INTEGER	[ OUT ]	language_int:	INTEGER	[ IN ]
err_num:	INTEGER	[ IN ]											
err_sev:	INTEGER	[ OUT ]											
err_flags:	INTEGER	[ OUT ]											
language_int:	INTEGER	[ IN ]											
<b>Comments:</b>	<p><i>err_num</i> is the error number whose message is looked for  <i>err_sev</i> is the error severity  <i>err_flags</i> 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.  <i>language_int</i> is the specified language:</p> <ul style="list-style-type: none"> <li>• 0 = default</li> <li>• 1 = English</li> <li>• 2 = French</li> <li>• 3 = German</li> <li>• 4 = Italian</li> <li>• 5 = Portuguese</li> <li>• 6 = Spanish</li> <li>• 7 = Turkish</li> </ul>												

- 8 = Chinese.

## 11.45 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

## 11.46 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 flib NOHOLD
VAR s : STRING[30]
ROUTINE filefound(as_name : STRING) : BOOLEAN EXPORTED FROM flib
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 flib

```

**See also:**    [ERR\\_TRAP\\_ON Built-In Procedure](#)

## 11.47 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 40075) 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

**Examples:**

```

PROGRAM flib NOHOLD
VAR s : STRING[30]
ROUTINE filefound(as_name : STRING) : BOOLEAN EXPORTED FROM flib
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 flib

```

## 11.48 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
```

## 11.49 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
```

## 11.50 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)
    -- 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
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

```

## 11.51 FL\_SET\_POS Built-In Procedure

The FL\_SET\_POS built-in procedure sets the file position for the next read/write operation.

<b>Calling Sequence:</b>	FL_SET_POS (lun_id, file_pos)				
<b>Parameters:</b>	<table border="0"> <tr> <td>lun_id : INTEGER</td> <td>[IN]</td> </tr> <tr> <td>file_pos : INTEGER</td> <td>[IN]</td> </tr> </table>	lun_id : INTEGER	[IN]	file_pos : INTEGER	[IN]
lun_id : INTEGER	[IN]				
file_pos : INTEGER	[IN]				
<b>Comments:</b>	<p>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.  <i>lun_id</i> can be any user-defined INTEGER variable that represents an open  logical unit number (LUN).  <i>lun_id</i> must be opened for random access and read or an error results.</p>				
<b>Examples:</b>	<pre> 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) </pre>				

```

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

```

## 11.52 FL\_STATE Built-In Function

The FL\_STATE built-in function obtains information about a file.

<b>Calling Sequence:</b>	FL_STATE (file_name_str, < size_int <, time_int <, attr_int >>>)								
<b>Return Type:</b>	BOOLEAN								
<b>Parameters:</b>	<table border="0"> <tbody> <tr> <td>file_name_string : STRING</td><td>[IN]</td></tr> <tr> <td>size_int : INTEGER</td><td>[OUT]</td></tr> <tr> <td>time_int : INTEGER</td><td>[OUT]</td></tr> <tr> <td>attr_int : INTEGER</td><td>[OUT]</td></tr> </tbody> </table>	file_name_string : STRING	[IN]	size_int : INTEGER	[OUT]	time_int : INTEGER	[OUT]	attr_int : INTEGER	[OUT]
file_name_string : STRING	[IN]								
size_int : INTEGER	[OUT]								
time_int : INTEGER	[OUT]								
attr_int : INTEGER	[OUT]								
<b>Comments:</b>	<p>The return value indicates whether the file exists or not.  <i>file_name_string</i> identifies the file whose information is to be obtained.  <i>size_int</i> will be set to the size of the file in bytes.  <i>time_int</i> will be set to the time of the file creation.  <i>attr_int</i> 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 <i>attr_int</i>:</p> <ul style="list-style-type: none"> <li>- Bit 1 : read_only file</li> <li>- Bit 2 : hidden file</li> <li>- Bit 3 : system file</li> <li>- Bit 5: directory</li> <li>- Bit 6: folder (.ZIP)</li> </ul> <p>As an example, if a file is hidden and read_only, the value of <i>attr_int</i> will be 3 (bits 1 and 2 are set).</p>								
<b>Examples:</b>	<pre> exist_bool := FL_STATE(file_name_str) exist_bool := FL_STATE(file_name_str, size_int, time_int) </pre>								

## 11.53 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:** port : INTEGER [IN]  
flow\_tbl\_idx: INTEGER [IN]

**Comments:** *port* is an INTEGER port reference (e.g. \$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](#)

## 11.54 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](#)

## 11.55 HDIN\_READ Built-In Procedure

The HDIN\_READ built-in procedure reads the positional value saved upon a high speed digital input (\$HDIN) trigger.

**Calling Sequence:** HDIN\_READ (pos\_class <, arm\_num>)

**Parameters:** pos\_class : || POSITION | JOINTPOS | XTNDPOS || [OUT]  
arm\_num : INTEGER [IN]

- Comments:** *pos\_class* indicates a POSITION, JOINTPOS, or XTNDPOS variable.  
*pos\_class* is set to the saved positional value.  
*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:** See the example program described in [HDIN\\_SET Built-In Procedure](#).

## 11.56 HDIN\_SET Built-In Procedure

The HDIN\_SET Built-In procedure sets a high speed digital input (\$HDIN).

**Calling Sequence:** HDIN\_SET (servo, enbl, cont, record\_flag <, arm\_num>...)

<b>Parameters:</b>	servo:	INTEGER	[ IN]
	enbl:	BOOLEAN	[ IN]
	cont:	BOOLEAN	[ IN]
	record_flag:	BOOLEAN	[ IN]
	arm_num:	INTEGER	[ IN]

- Comments:** When a high speed digital input is set, the arm can be stopped and its current position recorded.  
*servo* indicates the number of the associated DSA module.  
*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.  
*arm\_num* indicates which arms, if any, are to be stopped. Up to four arm numbers can be specified as separate parameters. The arms must be on the specified servo board.  
 The HDIN interrupt is triggered by a falling edge (positive to negative). For detecting the negative transition of the HDIN, events from 121 to 124 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 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 [Predefined Variables List](#) chapter.

- Examples:** 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)
        -- 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)
        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
                $DSA_DATA[1].HDIN_SUSP:=1
            WHEN START DO      -- reenable the HDIN triggering when motion
                -- restarts (START button)
                $DSA_DATA[1].HDIN_SUSP:= 0
        ENDCONDITION
        CONDITION[2]:
        WHEN EVENT 121 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
            $DSA_DATA[1].HDIN_SUSP :=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)
        $DSA_DATA[1].HDIN_SUSP := 1
        -- pnt0004j is the first point of research of the workpiece
        MOVE LINEAR TO pnt0004j WITH CONDITION[3]
        ....
    END hdin_ex

```

## 11.57 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

## 11.58 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
ROUTINE f1 (VAR spd: 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 WITH $PAR = f1(10)
  ...
END prg_1

```

## 11.59 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> )

<b>Parameters:</b>	jnt_var: JOINTPOS [ OUT]
ax1: REAL [ IN]	
ax2: REAL [ IN]	
ax3: REAL [ IN]	
ax4: REAL [ IN]	

ax5:	REAL	[ IN ]
ax6:	REAL	[ IN ]
ax7:	REAL	[ IN ]
ax8:	REAL	[ IN ]
ax9:	REAL	[ IN ]
ax10:	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

**Examples:** JNT (pnt0002j, 10.2, 0, 0, 5.7)

## 11.60 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*)

<b>Parameters:</b>	<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

```

## 11.61 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.

<b>Calling Sequence:</b>	JNTP_TO_POS ( <i>jnt_expr</i> , <i>pos_var</i> <, <i>base_ref</i> , <i>tool_ref</i> , <i>ufr_ref</i> , <i>dyn_flg</i> >)																		
<b>Parameters:</b>	<table border="0"> <tbody> <tr> <td><i>jnt_expr</i> :</td> <td>JOINTPOS</td> <td>[ IN ]</td> </tr> <tr> <td><i>pos_var</i> :</td> <td>POSITION</td> <td>[ OUT ]</td> </tr> <tr> <td><i>base_ref</i> :</td> <td>POSITION</td> <td>[ IN ]</td> </tr> <tr> <td><i>tool_ref</i> :</td> <td>POSITION</td> <td>[ IN ]</td> </tr> <tr> <td><i>ufr_ref</i> :</td> <td>POSITION</td> <td>[ IN ]</td> </tr> <tr> <td><i>dyn_flg</i> :</td> <td>BOOLEAN</td> <td>[ IN ]</td> </tr> </tbody> </table>	<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 ]
<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 ]																	
<b>Comments:</b>	<p><i>jnt_expr</i> is the JOINTPOS expression to be converted.  <i>pos_var</i> is the POSITION variable that gets set to the result of the conversion.  <i>base_ref</i> is the \$BASE to be used while converting to POSITION  <i>tool_ref</i> is the \$TOOL to be used while converting to POSITION  <i>ufr_ref</i> is the \$UFRAME to be used while converting to POSITION  <i>dyn_flg</i> 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.</p>																		
	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.																		
<b>Examples:</b>	<pre> 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 </pre>																		
<b>See also:</b>	<a href="#">POS_TO_JNTP Built-In Procedure</a>																		

## 11.62 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.

<b>Calling Sequence:</b>	KEY_LOCK ( <i>physical_device</i> , <i>enabl_disabl</i> <, <i>flags</i> >)
--------------------------	--

<b>Parameters:</b>	physical_device : INTEGER enabl_disabl : BOOLEAN flags : INTEGER	[ IN] [ IN] [ IN]
<b>Comments:</b>	<p><i>physical_device</i> 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.</p> <p><i>enabl_disabl</i> specifies whether the device(s) are to be enabled (TRUE) or disabled (FALSE).</p> <p><i>flags</i> 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 <a href="#">Predefined Variables List</a> chapter for the possible values.</p> <p>An error occurs if the EXECUTE, EDIT, or TEACH environments are active or the state selector is set to REMOTE or AUTO.</p> <p>The device is automatically unlocked when the system enters the PROGR state, power failure recovery is initiated, or a fatal error is detected.</p>	
<b>Examples:</b>	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	

## 11.63 LN Built-In Function

The LN built-in function returns the natural logarithm of a number.

<b>Calling Sequence:</b>	LN (number)
<b>Return Type:</b>	REAL
<b>Parameters:</b>	number : REAL
<b>Comments:</b>	number must be greater than zero or an error results.
<b>Examples:</b>	X := LN(5) -- x = 1.61 X := LN(62.4) -- x = 4.13356 X := LN(angle - offset)

## 11.64 MEM\_SPACE Built-In Procedure

MEM\_SPACE built-in procedure returns information about space in C4G memory.

<b>Calling Sequence:</b>	MEM_SPACE (total, num_blocks, largest)
<b>Parameters:</b>	total : INTEGER num_blocks : INTEGER largest : INTEGER
<b>Comments:</b>	<p><i>total</i> is the total number of free bytes available.</p> <p><i>num_blocks</i> is the number of individual free blocks.</p> <p><i>largest</i> is the size, in bytes, of the largest block available.</p>

**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)
```

## 11.65 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:**            *path\_var* is the PATH variable to be modified.  
*num\_nodes* 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.

**Examples:**

```
NODE_APP( weld_pth)                -- appends one node
NODE_APP( weld_pth, 4)                -- appends four nodes
```

## 11.66 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:**            *path\_var* is the PATH variable to be modified.  
*node\_idx* 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.  
*num\_nodes* is the number of nodes to delete from the path, starting with *node\_idx*. 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.

**Examples:**

```
ROUTINE clear_path (name : PATH OF weld_node)
BEGIN
  NODE_DEL(name, 1, PATHLEN(name))
END clear_path
```

## 11.67 NODE\_GET\_NAME Built-In Procedure

The NODE\_GET\_NAME built-in procedure obtains the symbolic name of a path node.

<b>Calling Sequence:</b>	NODE_GET_NAME (path_var, node_num, name)		
<b>Parameters:</b>	path_var	: PATH	[ IN ]
	node_num	: INTEGER	[ IN ]
	name	: STRING	[ OUT ]
<b>Comments:</b>	<p><i>path_var</i> is the PATH variable for the node.                  The node name <i>node_num</i> is obtained.  <i>name</i> will be set to the symbolic name of <i>node_num</i> node of <i>path_var</i> path. If the symbolic name is longer than the maximum length of name, it will be truncated.                  An error occurs if node <i>node_num</i> does not have a symbolic name.                  An error occurs if <i>node_num</i> is less than 1 or greater than the current length of <i>path_var</i>.                  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.</p>		
<b>Examples:</b>	NODE_GET_NAME( weld_pth, 3, str_var)		

## 11.68 NODE\_INS Built-In Procedure

The NODE\_INS built-in procedure inserts uninitialized nodes into a PATH variable.

<b>Calling Sequence:</b>	NODE_INS (path_var, node_idx <, num_nodes>)		
<b>Parameters:</b>	path_var	: PATH	[ IN, OUT ]
	node_idx	: INTEGER	[ IN ]
	num_nodes	: INTEGER	[ IN ]
<b>Comments:</b>	<p><i>path_var</i> is the PATH variable to be modified.                  The new nodes will be inserted after node <i>node_idx</i>. <i>node_idx</i> must be in the range from 0 to the total number of nodes in the path.  <i>num_nodes</i> is the number of nodes to insert into the path. If not specified, one node is inserted.                  All nodes from <i>node_idx</i>+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 <i>node_idx</i> 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.</p>		
<b>Examples:</b>	NODE_INS( weld_pth, 3) -- inserts one node after node 3 NODE_INS( weld_pth, 3, 4) -- inserts four nodes after node 3		

## 11.69 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>)

<b>Parameters:</b>	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)

## 11.70 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 it 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>)

<b>Parameters:</b>	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 (e.g. \$WORD).

*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
ENCONDITION
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]
$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 := TO 5 DO
  $ON_POS_TBL[i].OP_TOOL := $TOOL
  $ON_POS_TBL[i].OP_UFRAME := $UFRAME
  $ON_POS_TBL[i].OP_TOOL_DISBL := 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](#).

## 11.71 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 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.

<b>Calling Sequence:</b>	ON_JNT_SET_DIG (port, on_pos_idx, jnt_pos <, jnt_mask>)												
<b>Parameters:</b>	<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 ]											
<b>Comments:</b>	<p><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 digital port.  <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.  <i>jnt_mask</i> 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. <i>jnt_mask</i> is an optional parameter and, if not specified, it assumes the default value of \$ARM_DATA[arm_num].JNT_MASK.</p>												
<b>Examples:</b>	See the example of ON_JNT_SET.												
<b>See also:</b>	<a href="#">ON_POS Built-In Procedure</a> , <a href="#">ON_JNT_SET Built-In Procedure</a> and <a href="#">\$ON_POS_TBL: ON POS table data</a> .												

## 11.72 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.

<b>Calling Sequence:</b>	ON_POS (flag, on_pos_idx<, arm_num>)									
<b>Parameters:</b>	<table border="0"> <tr> <td>flag</td> <td>: BOOLEAN</td> <td>[ IN ]</td> </tr> <tr> <td>on_pos_idx</td> <td>: INTEGER</td> <td>[ IN ]</td> </tr> <tr> <td>arm_num</td> <td>: INTEGER</td> <td>[ IN ]</td> </tr> </table>	flag	: BOOLEAN	[ IN ]	on_pos_idx	: INTEGER	[ IN ]	arm_num	: INTEGER	[ IN ]
flag	: BOOLEAN	[ IN ]								
on_pos_idx	: INTEGER	[ IN ]								
arm_num	: INTEGER	[ IN ]								
<b>Comments:</b>	<p><i>flag</i> is a BOOLEAN value indicating if the algorithm should be enabled (ON) or disabled (OFF).  <i>on_pos_idx</i> is the \$ON_POS_TBL index of the element to which the ON POS feature should apply.  <i>arm_num</i> 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.</p>									

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.

## 11.73 ON\_POS\_SET Built-In Procedure

The ON\_POS\_SET 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.

<b>Calling Sequence:</b>	ON_POS_SET (port, bit, on_pos_idx)	
<b>Parameters:</b>	port : INTEGER	[ IN ]
	bit : INTEGER	[ IN ]
	on_pos_idx : INTEGER	[ IN ]
<b>Comments:</b>	<p><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.</p>	
<b>Examples:</b>	See example associated to <a href="#">ON_POS Built-In Procedure</a>	
<b>See also:</b>	<a href="#">ON_POS Built-In Procedure</a> , <a href="#">\$ON_POS_TBL: ON POS table data</a> and On Position Feature in Motion Programming Manual	

## 11.74 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.

<b>Calling Sequence:</b>	ON_POS_SET (port, bit, on_pos_idx)	
<b>Parameters:</b>	port : BOOLEAN	[ IN ]
	on_pos_idx : INTEGER	[ IN ]
<b>Comments:</b>	<p><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.</p>	
<b>Examples:</b>	ON_POS_SET_DIG (\$DOUT[5],3)	
<b>See also:</b>	<a href="#">ON_POS_SET Built-In Procedure</a> , <a href="#">ON_POS Built-In Procedure</a> , <a href="#">\$ON_POS_TBL: ON POS table data</a> and On Position Feature in Motion Programming Manual	

## 11.75 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>)`

<b>Parameters:</b>	<code>port</code> : INTEGER	[ IN ]
	<code>bit</code> : INTEGER	[ IN ]
	<code>arm_num</code> : INTEGER	[ IN ]

**Comments:**  
`port` is an INTEGER port reference (e.g. \$WORD).  
`bit` is an INTEGER expression indicating the bit associated to the \$CRNT\_DATA[arm\_num].OT\_COARSE value.  
`arm_num` 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

## 11.76 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>)`

<b>Parameters:</b>	<code>port</code> : BOOLEAN	[ IN ]
	<code>arm_num</code> : 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

## 11.77 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

```

## 11.78 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
```

## 11.79 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
```

## 11.80 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
```

## 11.81 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.

<b>Calling Sequence:</b>	POS_COMP_IDL (pos_comp, pos_idl)	
<b>Parameters:</b>	pos_comp :    POSITION   JOINTPOS   XTNDPOS	[ IN ]
	pos_idl :    POSITION   JOINTPOS   XTNDPOS	[ OUT ]
<b>Comments:</b>	<i>pos_comp</i> is the compensated position. <i>pos_idl</i> is the ideal position that is returned in output after the procedure call.	
<b>Examples:</b>	<pre> 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 </pre>	
<b>See also:</b>	<a href="#">POS_IDL_COMP Built-In Procedure</a>	

## 11.82 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).

<b>Calling Sequence:</b>	POS_CORRECTION (initial_pos, final_pos, angleX, angleY, angleZ, frame <, arm>)	
<b>Parameters:</b>	initial_pos : POSITION	[ IN ]
	final_pos : POSITION	[ OUT ]
	angleX : REAL	[ IN ]
	angleY : REAL	[ IN ]
	angleZ : REAL	[ IN ]
	frame : INTEGER	[ IN ]
	arm : INTEGER	[ IN ]
<b>Comments:</b>	<i>initial_pos</i> is the initial position <i>final_pos</i> is the calculated final position <i>angleX</i> is the angular variation around X <i>angleY</i> is the angular variation around Y <i>angleZ</i> is the angular variation around Z <i>frame</i> is the frame specification (BASE, TOOL, UFRAME) <i>arm</i> is the optional arm number.	

## 11.83 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*)

## 11.84 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
  
```

## 11.85 POS\_GET\_CNFG Built-In Function

The POS\_GET\_CNFG built-in function returns the configuration string of a specified

POSITION variable.

<b>Calling Sequence:</b>	POS_GET_CNFG (source_pos)
<b>Return Type:</b>	STRING
<b>Parameters:</b>	source_pos : POSITION [ IN]
<b>Comments:</b>	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 <a href="#">Data Representation</a> chapter for a description of the configuration string.
<b>Examples:</b>	<code>cfg_str := POS_GET_CNFG( cur_pos )</code>

## 11.86 POS\_GET\_LOC Built-In Function

The POS\_GET\_LOC built-in function returns the location vector of a specified position.

<b>Calling Sequence:</b>	POS_GET_LOC (source_pos)
<b>Return Type:</b>	VECTOR
<b>Parameters:</b>	source_pos : POSITION [ IN]
<b>Comments:</b>	The returned location vector represents the x, y, z components of source_pos.
<b>Examples:</b>	<pre>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</pre>

## 11.87 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.

<b>Calling Sequence:</b>	POS_GET_NORM (source_pos)
<b>Return Type:</b>	VECTOR
<b>Parameters:</b>	source_pos : POSITION [ IN]
<b>Comments:</b>	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

```

## 11.88 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

```

## 11.89 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.

## 11.90 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.

<b>Calling Sequence:</b>	POS_IDL_COMP (pos_idl, pos_comp)	
<b>Parameters:</b>	pos_idl :    POSITION   JOINTPOS   XTNDPOS	[ IN ]
	pos_comp :    POSITION   JOINTPOS   XTNDPOS	[ OUT ]
<b>Comments:</b>	<p><i>pos_idl</i> is the ideal position  <i>pos_comp</i> is the compensated position that is returned in output after the procedure call.</p>	
<b>Examples:</b>	<pre>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</pre>	
<b>See also:</b>	<a href="#">POS_COMP_IDL Built-In Procedure</a>	

## 11.91 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.

<b>Calling Sequence:</b>	POS_IN_RANGE (test_pos, bool_ans <, arm_num> <, base_ref, tool_ref, ufr_ref, dyn_flg>)	
<b>Parameters:</b>	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 ]
<b>Comments:</b>	<p><i>test_pos</i> is the position to be tested, to know whether it is in the range of the specified Arm.  <i>bool_ans</i> is set to TRUE if the arm can reach <i>test_pos</i> 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 <i>test_pos</i>.  If <i>arm_num</i> is not specified, the default arm is used.  <i>base_ref</i> is the \$BASE to be used while testing  <i>tool_ref</i> is the \$TOOL to be used while testing  <i>ufr_ref</i> is the \$UFRAME to be used while testing  <i>dyn_flg</i> is a flag to indicate whether or not dynamic references (such as</p>	

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

```

## 11.92 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*

## 11.93 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*)

## 11.94 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
    
```

## 11.95 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\_cfg*. Refer to [Data Representation](#) chapter for a description of the configuration string.

**Examples:** POS\_SET\_CNFG(*posn*, SW)

## 11.96 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
  
```

## 11.97 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
  
```

## 11.98 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.

<b>Calling Sequence:</b>	POS_SET_ORNT (posn, new_orient)
<b>Parameters:</b>	posn : POSITION [ IN, OUT] new_orient : VECTOR [ IN]
<b>Comments:</b>	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.
<b>Examples:</b>	<pre> 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 </pre>

## 11.99 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.

<b>Calling Sequence:</b>	POS_SET_RPY (pos_var, roll, pitch, yaw)
<b>Parameters:</b>	pos_var : POSITION [ IN] roll : REAL [ IN] pitch : REAL [ IN] yaw : REAL [ IN]
<b>Comments:</b>	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.

## 11.100 POS\_SHIFT Built-In Procedure

The POS\_SHIFT built-in procedure shifts a Cartesian position by a specified VECTOR.

<b>Calling Sequence:</b>	POS_SHIFT (posn, shf_vec)
<b>Parameters:</b>	posn : POSITION [ IN, OUT] shf_vec : VECTOR [ IN]
<b>Comments:</b>	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
```

## 11.101 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:**

<i>pos_expr</i> : POSITION	[ IN ]
<i>jnt_var</i> : JOINTPOS	[ 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:**

*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, 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
```

**See also:**

[JNTP\\_TO\\_POS Built-In Procedure](#)

## 11.102 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:** *posn* is the POSITION from which the components are to be extracted. The Cartesian coordinates of *posn* are assigned to *x*, *y*, and *z*. The Euler angles of *posn* are assigned to *e1*, *e2*, and *e3*. The configuration string of *posn* is assigned to *cnfg*.

**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

```

## 11.103 PROG\_OWNER Built-In Function

The PROG\_OWNER built-in function returns the code number of the EZ program that owns this routine.

**Calling Sequence:** PROG\_OWNER

**Return Type:** INTEGER

**Parameters:** None

**Comments:** The returned value is the code of the EZ program that owns the Built-In. If the program isn't an EZ program, the Built-In returns -1; if it is an EZ program but it isn't possible to find any number in the program name, the Built-In returns -3; if the conversion fails the Built-In returns -2.

**Examples:**

```

Prog_20 EZ
ROUTINE seam (i : INTEGER) EXPORTED FROM wa_appl
ROUTINE call_prog_20 EXPORTED FROM prog_20
BEGIN
    seam (PROG_OWNER) -- the PROG_OWNER Built-In returns
20
    END call_prog_20
BEGIN
    call_prog_20
END call_prog_20

```

## 11.104 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:**

<code>prog_name</code> : STRING	[ IN]
<code>line_num</code> : INTEGER	[ IN]
<code>rout_name</code> : STRING	[ IN]
<code>ext_prog_name</code> : STRING	[ IN]

**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 number of the being executed line

`rout_name` is the name of the being executed routine

`ext_prog_name` is the 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
  
```

## 11.105 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.

## 11.106 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:**

<code>ROUND(17.4)</code>	-- result is 17
<code>ROUND(96.5)</code>	-- result is 97
<code>ROUND(-17.4)</code>	-- result is -17
<code>ROUND(-96.5)</code>	-- result is -97

## 11.107 RPLC\_GET\_IDX Built-In Procedure

The RPLC\_GET\_IDX built-in procedure allows the user to obtain the resource index of a PLC project, and reference it when accessing to [\\$RPLC\\_DATA: Data of PLC resources](#) and [\\$RPLC\\_STS: Status of PLC resources](#) predefined variables.

**Calling Sequence:** `RPLC_GET_IDX(idx,resource_name,<configuration_name>,<project_name>)`

**Parameters:**

```
idx : INTEGER [OUT]
resource_name: STRING [IN]
configuration_name : STRING [IN]
project_name : STRING [IN]
```

**Comments:**

*idx* is the index of the resource  
*resource\_name* is the name of the resource  
*configuration\_name* is the name of the configuration  
*project\_name* is the name of the project

If *configuration\_name* and/or *project\_name* are not specified, they will be looked for in UD:\RESPLC directory.

Note that *configuration\_name* and *project\_name* are referred to the PLC software environment. Such information can be read by means of *ProgramResplcUtilityView* command.

**Examples:**

```
RPLC_GET_IDX (i,'res1', 'conf2', 'proj1')
```

## 11.108 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 WinC4G 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
```

## 11.109 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>>)
```

<b>Parameters:</b>	dev_num : INTEGER code : INTEGER scrn_num : INTEGER	[ IN] [ IN] [ IN]
<b>Comments:</b>	<p><i>dev_num</i> indicates the device containing the screen that is to be cleared. The following predefined constants represent devices that can be used for <i>dev_num</i>:</p> <p>PDV_TP -- Teach Pendant          PDV_CRT -- PC screen (Terminal Window), when using WinC4G Program</p> <p><i>code</i> is an INTEGER expression indicating what to clear from the user screen.</p> <p>The following predefined constants can be used:</p> <p>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</p> <p>If <i>code</i> is not specified, SCRN_CLR_Rem is used as the default.</p> <p><i>scrn_num</i> indicates the user-created screen to be cleared. If not specified, the system created user screen (SCRN_USER) is cleared.</p> <p>The SCRN_CREATE built-in function can be used to create new user screens.</p>	
<b>Examples:</b>	<pre>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</pre>	

## 11.110 SCRN\_CREATE Built-In Function

The SCRN\_CREATE built-in function creates a new user screen.

<b>Calling Sequence:</b>	SCRN_CREATE (scrn_name, dev_num)
<b>Return Type:</b>	INTEGER
<b>Parameters:</b>	scrn_name : STRING [ IN] dev_num : INTEGER [ IN]
<b>Comments:</b>	<p><i>scrn_name</i> is the name given to the new screen.</p> <p><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> <p>PDV_TP -- Teach Pendant          PDV_CRT -- PC screen (Terminal Window), when using WinC4G Program</p> <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.</p> <p>An error occurs if a screen called <i>scrn_name</i> already exists.</p>
<b>Examples:</b>	<pre>appl_screen := SCRN_CREATE(weld_disp, PDV_TP) SCRN_ADD(PDV_TP, appl_screen) IF SCRN_GET(PDV_TP) &lt;&gt; appl_screen THEN     SCRN_SET(PDV_TP, appl_screen) -- want appl screen on TP ENDIF</pre>

## 11.111 SCRN\_DEL Built-In Procedure

The SCRN\_DEL built-in procedure deletes a user-created screen.

**Calling Sequence:** SCRN\_DEL (scrn\_name)

**Parameters:** scrn\_name : STRING [ IN ]

**Comments:** scrn\_name indicates the screen to be deleted.

The screen must be removed from any SCRN key cycles before deletion. Use the SCRN\_REMOVE built-in for this purpose.

An error occurs if scrn\_name is not a defined screen or is currently in the cycle of a device SCRN key.

**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);
```

## 11.112 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 (dev\_num <, win\_name>)

**Return Type:** INTEGER

**Parameters:** dev\_num : INTEGER [ IN ]
 win\_name : 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 WinC4G 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
```

## 11.113 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 WinC4G 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);

```

## 11.114 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:** *dev\_num* 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 WinC4G Program  
*scrn\_num* 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 *scrn\_num* and the editor is not active, an error will occur.  
An error occurs if *scrn\_num* does not exist.

**Examples:**

```

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

```

## 11.115 SENSOR\_GET\_DATA Built-In Procedure

The SENSOR\_GET\_DATA will obtain sensor data from integrated sensors.

**Calling Sequence:**      `SENSOR_GET_DATA (sens_read, <,flag> <,arm>)`

**Parameters:**

<code>sens_read</code>	: ARRAY [ 6 ] of REAL	[ OUT ]
<code>flag</code>	: INTEGER	[ OUT ]
<code>arm</code>	: INTEGER	[ OUT ]

**Comments:**      `sens_read` 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.

`flag` 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.  
`arm` is optional and, if present, indicates an arm other than the default arm on which getting the sensor data.

**Examples:**

```
$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
```

## 11.116 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:**      `SENSOR_GET_OFST (ofst_tot <,arm>)`

**Parameters:**

<code>ofst_tot</code>	: ARRAY [ 6 ] of REAL	[ OUT ]
<code>arm</code>	: 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
```

## 11.117 SENSOR\_SET\_DATA Built-In Procedure

The SENSOR\_SET\_DATA built-in procedure is used to send offsets to the motion environment.

<b>Calling Sequence:</b>	SENSOR_SET_DATA (err_track <,arm>)
<b>Parameters:</b>	<b>err_track</b> : ARRAY [6] of REAL [IN] <b>arm</b> : INTEGER [IN]
<b>Comments:</b>	<i>err_track</i> 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 <i>err_track</i> not used elements must be initialized to zero. <i>arm</i> is optional and if present indicates an arm other than the default arm on which to set sensor data.
<b>Examples:</b>	<pre> 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       </pre>

## 11.118 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.

<b>Calling Sequence:</b>	SENSOR_SET_OFST (pos<,arm_num>)
<b>Parameters:</b>	<b>pos</b> : POSITION [IN] <b>arm_num</b> : INTEGER [IN]
<b>Comments:</b>	<i>pos</i> 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 <i>pos</i> parameter. <i>arm_num</i> indicates the arm number. If not specified, the \$PROG_ARM is used.
<b>Examples:</b>	\$SENSOR_SET_OFST (pos_var, 2)

## 11.119 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:**  
 bool is either TRUE or FALSE and indicates which state should be enabled for the sensor tracking.  
 arm\_num is optional and if present indicates an arm other than the default arm on which to get the sensor data.  
 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.  
 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.

SENSOR\_TRK mode is automatically disabled when the program that enabled it is deactivated.

**Examples:**

```

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
    $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
    ....
  
```

## 11.120 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

```

## 11.121 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

```

## 11.122 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

```

## 11.123 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 conc
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 conc
```

## 11.124 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

## 11.125 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.

## 11.126 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 = 'abcghijkl'
    letters := 'abcdefghijkl' -- restore for next example
    STR_DEL(letters, 8, 5) -- letters = 'abcdefg'
END str

```

## 11.127 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)
```

## 11.128 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:**

<i>source_string</i> : STRING	[ IN ]
<i>start_index</i> : INTEGER	[ IN ]
<i>bytes_number</i> : INTEGER	[ IN ]
<i>integer_val</i> : 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.
```

## 11.129 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:**

<i>source_string</i> : STRING	[ IN ]
<i>start_index</i> : INTEGER	[ IN ]
<i>bytes_number</i> : INTEGER	[ IN ]
<i>real_val</i> : 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.`

## 11.130 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
```

## 11.131 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
  
```

## 11.132 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 (source\_string, find\_string <,integer\_search\_par>)

**Return Type:** INTEGER

**Parameters:**

source_string : STRING	[ IN ]
find_string : STRING	[ IN ]
integer_search_par : INTEGER	[ IN ]
integer_val : INTEGER	[ OUT ]

**Comments:** source\_string is the string in which find\_string is to be searched for.

find\_string is the string to be searched for; if find\_string is not found, zero is returned.

integer\_search\_par is an optional integer parameter which indicates the direction of searching and the index where to start searching.

**Examples:**

```

PROGRAM str
VAR
  start_idx : INTEGER
  letters : STRING[10]
BEGIN
  letters := 'abcdefghijkl'
  start_idx := STR_LOC(letters, 'd') -- start_idx = 4
  STR_DEL(letters, start_idx, 2) -- letters = 'abcfgij'
END str
  
```

## 11.133 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

```

## 11.134 STR\_SET\_INT Built-In Procedure

This routine sets an INTEGER value inside a STRING at a specified index and for a certain number of bytes.

**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 assigned to the *destination\_string* variable.  
*destination\_string* is the string which the integer value is to be assigned to.  
*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:**

```

p := 'pippo'
STR_SET_INT (105,p,3,1) -- set the third byte in string p.
-- p becomes 'piipo'.

```

## 11.135 STR\_SET\_REAL Built-In Procedure

This routine sets a REAL value inside a STRING at a specified index and for a certain number of bytes.

**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 assigned to the *destination\_string* variable.  
*destination\_string* is the string which the real value is to be assigned to.  
*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:**

```
p := 'pippo'
STR_SET_REAL (105,p,3,1) -- set the third byte in string p.
```

## 11.136 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(i90.0.0.2) returns 0x5A000002.

This is useful when configuring some of the system parameters that represent IP addresses but take integer values.

**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

## 11.137 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

```

## 11.138 SYS\_ADJUST Built-In Procedure

Reserved.

## 11.139 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 Winc4g program is active). For example, if the FILER VIEW command is being requested, 'FV' would be used.

Refer to "Use of C4G Controller 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 Winc4g 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\ippo.cod`, when inside a SYS\_CALL, should be written as follows:

`SYS_CALL ('FD', 'UD:\usr1\ippo.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
- CCRR -- CONFIGURE CNTRLER RESTART RELOAD
- CI -- CONFIGURE IO\_CONF
- DCS -- DISPLAY CLOSE SELECT
- FUAH -- FILER UTILITY ATTRIBUTE HIDDEN
- FUAR -- FILER UTILITY ATTRIBUTE READONLY
- FUAS -- FILER UTILITY ATTRIBUTE SYSTEM
- 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\_CNG can be set to 1.

**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

```

## 11.140 SYS\_SETUP Built-In Procedure

Reserved.

## 11.141 TABLE\_ADD Built-In Procedure

The TABLE\_ADD built-in procedure adds a table to the DATA page of the TP4i/WiTP 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 TP4i/WiTP.

**Calling Sequence:** TABLE\_ADD (table\_name, owning\_program<,file\_for\_saving>)

<b>Parameters:</b>	table_name: STRING [ IN] owning_program: STRING [ IN] file_for_saving: STRING [ IN]
<b>Comments:</b>	The parameter <i>table_name</i> 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 <i>owning_program</i> parameter is the name of the program that owns the table TYPEDEF declaration. The optional parameter <i>file_for_saving</i> can be specified if the table should be saved in a .VAR file different than the owning program.
<b>Examples:</b>	<pre>PROGRAM exampletable 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 exampletable</pre>
<b>See also:</b>	<a href="#">TABLE_DEL Built-In Procedure</a> . <a href="#">Use of C4G manual, DATA page section</a> <a href="#">par. 16.2 User table creation from DATA environment on page 16-1.</a>

## 11.142 TABLE\_DEL Built-In Procedure

The TABLE\_DEL built-in procedure removes the link of the table with the DATA page of TP4i/WiTP.

<b>Calling Sequence:</b>	TABLE_DEL (table_name)
<b>Parameters:</b>	table_name: STRING [ IN]
<b>Comments:</b>	The parameter <i>table_name</i> is the name of the TYPEDEF which defines the table elements.
<b>Examples:</b>	TABLE_DEL( 'userTable' )
<b>See also:</b>	<a href="#">TABLE_ADD Built-In Procedure</a> . <a href="#">Use of C4G manual, DATA page section</a> <a href="#">par. 16.2 User table creation from DATA environment on page 16-1.</a>

## 11.143 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
```

## 11.144 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
```

## 11.145 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.

## 11.146 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.

<b>Calling Sequence:</b>	VAR_UNINIT (var_ref)
<b>Return Type:</b>	BOOLEAN
<b>Parameters:</b>	var_ref : any variable reference [ IN ]
<b>Comments:</b>	<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>).          If <i>var_ref</i> has not been given a value, TRUE is returned. Otherwise, FALSE is returned.          If an ARRAY variable is used, it must be subscripted.          If a RECORD or NODE variable is used, it must have a field specification.          If a PATH variable is specified, the whole path is tested. A PATH is considered uninitialized if it has zero nodes.          If a PATH node is tested, a specific field must be specified.</p>
<b>Examples:</b>	<pre>PROGRAM testinit VAR     ok : BOOLEAN     count : INTEGER (0)     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</pre>

## 11.147 VEC Built-In Function

The VEC built-in function returns a VECTOR created from the three specified REAL components.

<b>Calling Sequence:</b>	VEC (x, y, z)
<b>Return Type:</b>	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

## 11.148 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*)

## 11.149 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 Winc4g 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
```

## 11.150 WIN\_CLEAR Built-In Procedure

The WIN\_CLEAR built-in procedure clears a specified window.

**Calling Sequence:**      WIN\_CLEAR (clear\_spec <, win\_name>)

<b>Parameters:</b>	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
  - (whenWinc4g 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:')

## 11.151 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 Winc4g 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

```

## 11.152 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:**

win_name	: STRING	[ IN]
dev_num	: INTEGER	[ IN]
attributes	: INTEGER	[ IN]
num_rows	: 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 WinC4G 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\_ON -- bolding
- WIN\_BLINK\_ON -- 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)

## **11.153 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:')
```

## 11.154 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:**

<code>win_name</code>	:	STRING	[ IN ]
<code>scrn_num</code>	:	INTEGER	[ IN ]
<code>row_num</code>	:	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 Winc4g 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)

```

## 11.155 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:**

row	: INTEGER	[OUT]
col	: INTEGER	[OUT]
win_name	: 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 Winc4g 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
  
```

## 11.156 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:**

row	: INTEGER	[IN]
col	: INTEGER	[IN]
num_chars	: INTEGER	[IN]
win_name	: STRING	[IN]

**Comments:** *row* is the row position inside *win\_name*. If *row* is not specified or it has a negative value, the current row will be used.

*col* is the column position inside *win\_name*. If *col* is not specified or it has a negative value, the current column will be used.

*num\_chars* indicates the number of characters to be obtained. If *num\_chars* is not specified or it has a negative value, the entire row is obtained.

*win\_name* is the name of the window from which the characters are to be obtained. If not specified, the default window is used.

**Examples:**

```

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

```

## 11.157 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 Winc4g 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:**

file_name : STRING	[ IN ]
win_name : STRING	[ IN ]
start_row : INTEGER	[ IN ]
start_col : 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

```

## 11.158 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 Winc4g 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:')

```

## 11.159 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 Winc4g 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:')

```

## 11.160 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 Winc4g program is active), the created file will only be compatible with C4G systems that use Winc4g program Terminal opened.

**Calling Sequence:**      WIN\_SAVE ( *file\_name*, *win\_name* <, *start\_row* <, *start\_col* <, *num\_rows* <, *num\_cols* <, *output\_row* <, *output\_col*>>>> )

**Parameters:**

<i>file_name</i>	: STRING	[ IN ]
<i>win_name</i>	: STRING	[ IN ]
<i>start_row</i>	: INTEGER	[ IN ]
<i>start_col</i>	: INTEGER	[ IN ]
<i>num_rows</i>	: INTEGER	[ IN ]
<i>num_cols</i>	: INTEGER	[ IN ]
<i>output_row</i>	: INTEGER	[ IN ]
<i>output_col</i>	: 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

```

## 11.161 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, SCRNUSE is used.

**Examples:**

```

WIN_SEL ('menu:')
OPEN FILE lun ('CRT2:', 'RW')
WRITE lun ('Enter value: ')
WIN_SEL('CRT2:') -- on SCRNUSE
READ lun (val)
  
```

## 11.162 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 ( Winc4g 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
  
```

## 11.163 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 Winc4g 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:' )
```

## 11.164 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 Winc4g 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

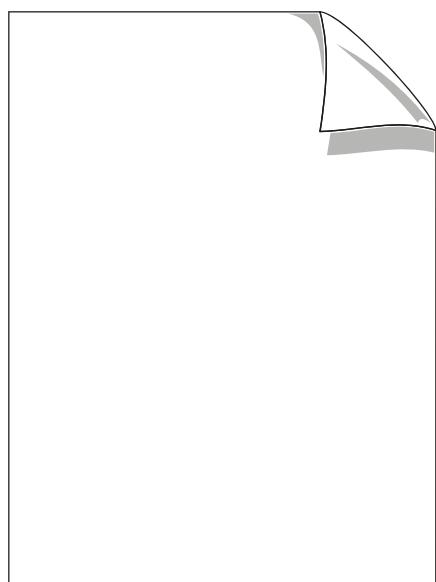
*parent* will be set to the name of the window upon which *win\_name* is popped up (if it isn't a fixed window).

*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

- 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 [INPUT/OUTPUT Port Arrays](#) 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.
- Field: the predefined variable is a field of the indicated predefined variable. For example field of arm\_data, field of crnt\_data, ecc.: Predefined variables belong to one of the following memory categories which indicates where the value is located:

## 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.

- DSA: these are variables related to the DSA 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 .C4G 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 .C4G 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 C4G Use and Maintenance Manuals.



**Note that complex structures such as \$ARM\_DATA, \$MCP\_DATA, \$DSA\_DATA are classified as “Not Saved” Load Category, even if some fields of theirs are always saved and loaded from .C4G 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.



**For further information see also Control Unit Use Manual - Chap. SYSTEM COMMANDS.**

## 12.3 Data Type

Predefined variables use the same data types as userdefined 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 (.C4G) is converted in an ASCII format (.PDL).

## 12.8 Predefined Variables groups

All predefined variables are classified in the following groups:

- [PLC System Variables](#)
- [PORT System Variables](#)
- [PROGRAM STACK System Variables](#)
- [ARM\\_DATA System Variables](#)
- [CRNT\\_DATA System Variables](#)
- [DSA\\_DATA System Variables](#)
- [MCP\\_DATA System Variables](#)
- [FBP\\_TBL System Variables](#)
- [WEAVE\\_TBL System Variables](#)
- [CONV\\_TBL System Variables](#)
- [ON\\_POS\\_TBL System Variables](#)
- [WITH MOVE System Variables](#)
- [WITH MOVE ALONG System Variables](#)
- [WITH OPEN FILE System Variables](#)
- [PATH NODE FIELD System Variables](#)
- [MISCELLANEOUS System Variables](#)

### 12.8.1 PLC System Variables

- \$RPLC\_DATA: Data of PLC resources
- \$RPLC\_STS: Status of PLC resources

### 12.8.2 PORT System Variables

- \$AIN: Analog input
- \$AOUT: Analog output
- \$BIT: PLC BIT data
- \$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
- \$GIN: Group input
- \$GOUT: Group output
- \$HDIN: High speed digital input
- \$IN: IN digital
- \$OUT: OUT digital
- \$PROG\_UBIT: Program user-defined bit memory
- \$PROG\_UBYTE: Program user-defined byte memory
- \$PROG ULONG: Program user-defined long word memory
- \$PROG\_UWORD: Program user-defined word memory
- \$SDIN: System digital input
- \$SDOUT: System digital output
- \$TIMER: Clock timer
- \$USER\_BIT: User-defined bit memory
- \$USER\_BYTE: User-defined byte memory
- \$USER\_LONG: User-defined long word memory
- \$USER\_WORD: User-defined word memory
- \$WORD: PLC WORD data

### 12.8.3 PROGRAM STACK System Variables

- \$CYCLE: Program cycle count
- \$ERROR: Last PDL2 Program Error
- \$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\_CNFG: Program configuration
- \$PROG\_CONDS: Defined conditions of a program
- \$PROG\_DEC\_OVR: Program deceleration override
- \$PROG\_NAME: Executing program name
- \$PROG\_SPD\_OVR: Program speed override
- \$PROG\_UADDR: Address of program user-defined memory access variables
- \$PROGULEN: Length of program memory access user-defined variables
- \$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\_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

#### 12.8.4 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: TP4i/WiTP AUX-A and AUX-B keys mapping
- \$AUX\_MASK: Auxiliary arm mask
- \$AUX\_OFST: Auxiliary axes offsets
- \$AUX\_SIK\_DRVON\_ENBL: Auxiliary axes provided of SIK
- \$AUX\_SIK\_MASK: Auxiliary axes provided of SIK
- \$AUX\_TYPE: Positioner type
- \$AX\_CNVRSN: Axes conversion
- \$AX\_INF: Axes inference
- \$AX\_LEN: Axes lengths
- \$AX\_OFST: Axes offsets
- \$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
- \$C4GOPEN\_JNT\_MASK: C4G Open Joint arm mask
- \$C4GOPEN\_MODE: C4G Open modality
- \$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
- \$CONV\_ACC\_LIM: Conveyor acceleration limit
- \$CONV\_BASE: Conveyor base frame
- \$CONV\_CNFG: Conveyor tracking configuration
- \$CONV\_SPD\_LIM: Conveyor speed limit
- \$CONV\_TYPE: Element in the Conveyor Table
- \$CONV\_WIN: Conveyor Windows
- \$CONV\_ZERO: Conveyor Position Transducer Zero
- \$DYN\_COLL\_FILTER: Dynamic Collision Filter
- \$DYN\_DELAY: Dynamic model delay
- \$DYN\_FILTER: Dynamic Filter
- \$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 carico
- \$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
- \$JNT\_LIMIT\_AREA: Joint limits of the work area
- \$JNT\_MASK: Joint arm mask
- \$JNT\_MTURN: Check joint Multi-turn
- \$JNT\_OVR: joint override
- \$JOGL\_INCR\_DIST: Increment Jog distance
- \$JOGL\_INCR\_ENBL: Jog incremental motion
- \$JOGL\_INCR\_ROT: Rotational jog increment
- \$JOGL\_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\_OVR: Run-time Linear speed override
- \$LOG\_TO\_DSA: Logical to physical DSA relationship
- \$LOG\_TO\_PHY: Logical to physical relationship
- \$MAN\_SCALE: Manual scale factor
- \$MCP\_BOARD: Motion Control Process board
- \$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
- \$SOP\_TOL\_DIST: On Pos-Jnt Tolerance distance
- \$SOP\_TOL\_ORNT: On Pos-Jnt Tolerance Orientation
- \$SOT\_TOL\_DIST: On Trajectory Tolerance distance
- \$SOT\_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\_STRESS\_PER: Maximum Stress allowed in Cartesian SmartMove4
- \$SM4\_SAT\_SCALE: SmartMove4 saturation thresholds
- \$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\_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\_FRICTION: Tool Friction
- \$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 CRNT\_DATA System Variables

- \$ARM\_DISB: Arm disable flags
- \$ARM\_SIMU: Arm simulate flag
- \$ARM\_SPACE: current Arm Space
- \$ARM\_VEL: Arm velocity
- \$CAUX\_POS: Cartesian positioner position
- \$COLL\_ENBL: Collision enabling flag
- \$CONV\_DIST: Conveyor shift in micron (mm/1000)
- \$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\_DEBUG: Cartesian Fly Debug
- \$FOLL\_ERR: Following error
- \$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)

## 12.8.6 DSA\_DATA System Variables

- \$D\_ALONG\_1D: Internal DSA data
- \$D\_AREAL\_1D: Internal DSA data

- \$D\_AXES: Internal DSA data
- \$D\_CTRL: Internal DSA data
- \$D\_HDIN\_SUSP: DSA\_DATA field for HDIN suspend
- \$D\_MTR: Internal DSA data

### 12.8.7 MCP\_DATA System Variables

- \$HDIN\_SUSP: HDIN Suspend
- \$IPERIOD: Interpolator period
- \$M\_ALONG\_1D: Internal motion control data
- \$REF\_ARMS: Reference arms

### 12.8.8 FBP\_TBL System Variables

- \$FB\_ADDR: Field Bus ADDR
- \$FB\_MA\_INIT: Field bus master init
- \$FB\_MA\_SLVS: Field bus master slaves init
- \$FB\_SLOT: field bus slot
- \$FB\_SL\_INIT: Field bus slave init
- \$FB\_TYPE: Field bus type

### 12.8.9 WEAVE\_TBL System Variables

- \$WV\_AMP\_PER: Weave amplitude percentage
- \$WV\_CNTR\_DLW: Weave center dwell
- \$WV\_END\_DLW: Weave end dwell
- \$WV\_LEFT\_AMP: Weave left amplitude
- \$WV\_LEFT\_DLW: Weave left dwell
- \$WV\_ONE\_CYCLE: Weave one cycle
- \$WV\_PLANE: Weave plane angle
- \$WV\_RIGHT\_AMP: Weave right amplitude
- \$WV\_RIGHT\_DLW: 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

### 12.8.10 CONV\_TBL System Variables

- \$CT\_JNT\_MASK: Conveyor Joint mask
- \$CT\_RADIUS: Conveyor radius in mm

- \$CT\_RES: Conveyor position in motor turns
- \$CT\_SCC: Conveyor SCC
- \$CT\_TX\_RATE: Transmission rate

### 12.8.11 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 Posposition 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

### 12.8.12 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
- \$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\_FRICTION: Tool Friction
- \$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: Weave amplitude percentage

### 12.8.13 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
- \$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

### 12.8.14 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

## 12.8.15 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

## 12.8.16 MISCELLANEOUS System Variables

- \$APPL\_ID: Application Identifier
- \$APPL\_NAME: Application Identifiers
- \$APPL\_OPTIONS: Application OptionsApplication Options
- \$ARM\_DATA: Robot arm data
- \$ARM\_USED: Program use of arms
- \$BACKUP\_SET: Default devices

- \$BOARD\_DATA: Board data
- \$BOOTLINES: Bootline read-only
- \$BREG: Boolean registers - saved
- \$BREG\_NS: Boolean registers - not saved
- \$B\_ASTR\_1D\_NS: Board string data
- \$B\_ALONG\_1D: Internal arm data
- \$B\_ALONG\_1D\_NS: Internal arm data
- \$B\_NVRAM: NVRAM data of the board
- \$CIO\_AIN: Configuration for AIN
- \$CIO\_AOUT: Configuration for AOUT
- \$CIO\_CAN: Configuration for Can Bus
- \$CIO\_CROSS: Configuration for I/O cross copying
- \$CIO\_DIN: Configuration for DIN
- \$CIO\_DOUT: Configuration for DOUT
- \$CIO\_FMI: Configuration for \$FMI
- \$CIO\_FMO: Configuration for \$FMO
- \$CIO\_GIN: Configuration for GIN
- \$CIO\_GOUT: Configuration for GOUT
- \$CIO\_IN\_APP: Configuration for IN
- \$CIO\_OUT\_APP: Configuration for OUT
- \$CIO\_SDIN: Configuration for the system digital inputs
- \$CIO\_SDOOUT: Configuration for the system digital outputs
- \$CIO\_SYS\_CAN: Configuration of the system modules on Can Bus
- \$CNTRL\_CNFG: Controller configuration mode
- \$CNTRL\_INIT: Controller initialization mode
- \$CNTRL\_OPTIONS: Controller Options
- \$CNTRL\_TZ: Controller Time Zone
- \$CONV\_TBL: Conveyor tracking table data
- \$CRNT\_DATA: Current Arm data
- \$DEPEND\_DIRS: Dependancy 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
- \$DSA\_DATA: DSA data
- \$DV\_STS: the status of DV4\_CNTRL call

- \$DV\_TOUT: Timeout for asynchronous DV4\_CNTRL calls
- \$D\_ALONG\_1D: Internal DSA data
- \$D\_AREAL\_1D: Internal DSA data
- \$D\_AXES: Internal DSA data
- \$D\_CTRL: Internal DSA data
- \$D\_MTR: Internal DSA data
- \$EMAIL\_INT: Email integer configuration:
- \$EMAIL\_STR: Email string configuration:
- \$EXE\_HELP: Help on Execute command
- \$FBP\_TBL: Field Bus Table data
- \$FB\_CNFG: Controller fieldbuses configuration mode
- \$FB\_INIT: Controller fieldbuses initialization mode
- \$FLOW\_TBL: Flow modulation algorithm table data
- \$FL\_DLMT: 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\_CNFG: Latched alarm configuration setting
- \$MCP\_DATA: Motion Control Process data

- \$MDM\_INT: Modem Configuration
- \$MDM\_STR: Modem Configuration:
- \$NET\_B: Ethernet Boot Setup
- \$NET\_B\_DIR: Ethernet Boot Setup Directory
- \$NET\_C\_CNFG: 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\_DEVICES: Number of devices
- \$NUM\_DSAS: Number of DSAs
- \$NUM\_LUNS: Number of LUNs
- \$NUM\_MB: Number of motion buffers
- \$NUM\_MB\_AHEAD: Number of motion buffers ahead
- \$NUM\_MCPS: Number of Motion Control Process
- \$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
- \$PPP\_INT: PPP Configuration
- \$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
- \$SREG: String registers - saved
- \$SREG\_NS: String registers - not saved
- \$STARTUP: Startup program
- \$STARTUP\_USER: Startup user
- \$SWIM\_ADDR: SWIM address
- \$SWIM\_CNFG: SWIM configuration mode
- \$SWIM\_INIT: SWIM Board initialization parameters
- \$SYS\_INP\_MAP: Configuration for system bits in Input on fieldbuses
- \$SYS\_ERROR: Last system error
- \$SYS\_ID: Robot System identifier
- \$SYS\_OUT\_MAP: Configuration for system bits in Output on fieldbuses
- \$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
- \$USER\_ADDR: Address of user-defined variables
- \$USER\_LEN: Length of User-defined variables
- \$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

- \$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: TP4i/WiTP AUX-A and AUX-B keys mapping
- \$AUX\_MASK: Auxiliary arm mask
- \$AUX\_OFST: Auxiliary axes offsets
- \$AUX\_SIK\_DRVON\_ENBL: Auxiliary axes provided of SIK
- \$AUX\_SIK\_MASK: Auxiliary axes provided of SIK
- \$AUX\_TYPE: Positioner type
- \$AX\_CNVRSN: Axes conversion
- \$AX\_INF: Axes inference
- \$AX\_LEN: Axes lengths
- \$AX\_OFST: Axes offsets
- \$A\_ALONG\_1D: Internal arm data
- \$A\_ALONG\_2D: Internal arm data
- \$A\_AREAL\_1D: Internal arm data
- \$A\_AREAL\_2D: Internal arm data
- \$B\_ASTR\_1D\_NS: Board string data
- \$BACKUP\_SET: Default devices
- \$BASE: Base of arm
- \$BIT: PLC BIT data
- \$BOARD\_DATA: Board data

- \$BOOTLINES: Bootline read-only
- \$BREG: Boolean registers - saved
- \$BREG\_NS: Boolean registers - not saved
- \$B\_ALONG\_1D: Internal arm data
- \$B\_ALONG\_1D\_NS: Internal arm data
- \$B\_NVRAM: NVRAM data of the board
- \$C4GOPEN\_JNT\_MASK: C4G Open Joint arm mask
- \$C4GOPEN\_MODE: C4G Open modality
- \$C4G\_RULES: C4G Save & Load rules
- \$CAL\_DATA: Calibration data
- \$CAL\_SYS: System calibration position
- \$CAL\_USER: User calibration position
- \$CAUX\_POS: Cartesian positioner position
- \$CIO\_AIN: Configuration for AIN
- \$CIO\_AOUT: Configuration for AOUT
- \$CIO\_CAN: Configuration for Can Bus
- \$CIO\_CROSS: Configuration for I/O cross copying
- \$CIO\_DIN: Configuration for DIN
- \$CIO\_DOUT: Configuration for DOUT
- \$CIO\_FMI: Configuration for \$FMI
- \$CIO\_FMO: Configuration for \$FMO
- \$CIO\_GIN: Configuration for GIN
- \$CIO\_GOUT: Configuration for GOUT
- \$CIO\_IN\_APP: Configuration for IN
- \$CIO\_OUT\_APP: Configuration for OUT
- \$CIO\_SDIN: Configuration for the system digital inputs
- \$CIO\_SDOOUT: Configuration for the system digital outputs
- \$CIO\_SYS\_CAN: Configuration of the system modules on Can Bus
- \$CNFG\_CARE: Configuration care
- \$CNTRL\_CNFG: Controller configuration mode
- \$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\_ACC\_LIM: Conveyor acceleration limit
- \$CONV\_BASE: Conveyor base frame
- \$CONV\_CNGF: Conveyor tracking configuration
- \$CONV\_DIST: Conveyor shift in micron (mm/1000)
- \$CONV\_SHIFT: Conveyor shift in mm
- \$CONV\_SPD: Conveyor speed
- \$CONV\_SPD\_LIM: Conveyor speed limit
- \$CONV\_TBL: Conveyor tracking table data
- \$CONV\_TYPE: Element in the Conveyor Table
- \$CONV\_WIN: Conveyor Windows
- \$CONV\_ZERO: Conveyor Position Transducer Zero
- \$CRNT\_DATA: Current Arm data
- \$CT\_JNT\_MASK: Conveyor Joint mask
- \$CT\_RADIUS: Conveyor radius in mm
- \$CT\_RES: Conveyor position in motor turns
- \$CT\_SCC: Conveyor SCC
- \$CT\_TX\_RATE: Transmission rate
- \$CUSTOM\_ARM\_ID: Identificator for the arm
- \$CUSTOM\_CNTRL\_ID: Identificator for the Controller
- \$CYCLE: Program cycle count
- \$C\_ALONG\_1D: Internal current arm data
- \$C\_AREAL\_1D: Internal current arm data
- \$C\_AREAL\_2D: Internal current arm data
- \$DEPEND\_DIRS: Dependancy 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
- \$DSA\_DATA: DSA data
- \$DV\_STS: the status of DV4\_CNTRL call
- \$DV\_TOUT: Timeout for asynchronous DV4\_CNTRL calls
- \$DYN\_COLL\_FILTER: Dynamic Collision Filter
- \$DYN\_DELAY: Dynamic model delay
- \$DYN\_FILTER: Dynamic Filter

- \$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 carico
- \$D\_LONG\_1D: Internal DSA data
- \$D\_AREAL\_1D: Internal DSA data
- \$D\_AXES: Internal DSA data
- \$D\_CTRL: Internal DSA data
- \$D\_HDIN\_SUSP: DSA\_DATA field for HDIN suspend
- \$D\_MTR: Internal DSA data
- \$EMAIL\_INT: Email integer configuration
- \$EMAIL\_STR: Email string configuration
- \$ERROR: Last PDL2 Program Error
- \$EXE\_HELP: Help on Execute command
- \$FBP\_TBL: Field Bus Table data
- \$FB\_ADDR: Field Bus ADDR
- \$FB\_CNFG: Controller fieldbuses configuration mode
- \$FB\_INIT: Controller fieldbuses initialization mode
- \$FB\_MA\_INIT: Field bus master init
- \$FB\_MA\_SLVS: Field bus master slaves init
- \$FB\_SLOT: field bus slot
- \$FB\_SL\_INIT: Field bus slave init
- \$FB\_TYPE: Field bus type
- \$FDIN: Functional digital input
- \$FDOUT: Functional digital output
- \$FL\_DLMT: Array of delimiters
- \$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
- \$GIN: Group input
- \$GOUT: Group output
- \$GUN: Electrical welding gun
- \$HAND\_TYPE: Type of hand
- \$HDIN: High speed digital input
- \$HDIN\_SUSP: HDIN Suspend
- \$HLD\_DEC\_PER: Hold deceleration percentage
- \$HOME: Arm home position
- \$IN: IN digital
- \$IPERIOD: Interpolator period
- \$IREG: Integer register - saved
- \$IREG\_NS: Integer registers - not saved
- \$JERK: Jerk control values
- \$JNT\_LIMIT\_AREA: Joint limits of the work area
- \$JNT\_MASK: Joint arm mask
- \$JNT\_MTURN: Check joint Multi-turn
- \$JNT\_OVR: joint override
- \$JOGL\_INCR\_DIST: Increment Jog distance
- \$JOGL\_INCR\_ENBL: Jog incremental motion
- \$JOGL\_INCR\_ROT: Rotational jog increment
- \$JOGL\_SPD\_OVR: Jog speed override

- \$JPAD\_DIST: Distance between user and Jpad
- \$JPAD\_ORNT: TP4i/WiTP Angle setting
- \$JPAD\_TYPE: TP4i/WiTP Jpad modality rotational or translational
- \$JREG: Jointpos registers - saved
- \$JREG\_NS: Jointpos register - not saved
- \$LATCH\_CNGF: 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\_OVR: Run-time Linear speed override
- \$LOG\_TO\_DSA: Logical to physical DSA relationship
- \$LOG\_TO\_PHY: Logical to physical relationship
- \$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
- \$MCP\_BOARD: Motion Control Process board
- \$MCP\_DATA: Motion Control Process data
- \$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
- \$M\_ALONG\_1D: Internal motion control data
- \$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\_DEVICES: Number of devices
- \$NUM\_DSAS: Number of DSAs
- \$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\_MCPS: Number of Motion Control Process
- \$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 Posposition 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
- \$PPP\_INT: PPP Configuration
- \$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\_CNFG: Program configuration
- \$PROG\_CONDS: Defined conditions of a program
- \$PROG\_DEC\_OVR: Program deceleration override
- \$PROG\_NAME: Executing program name
- \$PROG\_SPD\_OVR: Program speed override
- \$PROG\_UADDR: Address of program user-defined memory access variables
- \$PROG\_UBIT: Program user-defined bit memory
- \$PROG\_UBYTE: Program user-defined byte memory
- \$PROG\_ULEN: Length of program memory access user-defined variables
- \$PROG ULONG: Program user-defined long word memory
- \$PROG\_UWORD: Program user-defined word memory
- \$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
- \$REF\_ARMS: Reference arms
- \$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
- \$RPLC\_DATA: Data of PLC resources
- \$RPLC\_STS: Status of PLC resources
- \$RREG: Real registers - saved
- \$RREG\_NS: Real registers - not saved
- \$SAFE\_ENBL: Safe speed enabled
- \$SDIN: System digital input
- \$SDOUT: System digital output
- \$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
- \$SFRAME: Sensor frame of an arm
- \$SING\_CARE: Singularity care
- \$SM4C\_STRESS\_PER: Maximum Stress allowed in Cartesian SmartMove4
- \$SM4\_SAT\_SCALE: SmartMove4 saturation thresholds
- \$SPD\_OPT: Type of speed control
- \$SREG: String registers - saved
- \$SREG\_NS: String registers - not saved
- \$STARTUP: Startup program
- \$STARTUP\_USER: Startup user
- \$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
- \$SWIM\_ADDR: SWIM address
- \$SWIM\_CNFG: SWIM configuration mode
- \$SWIM\_INIT: SWIM Board initialization parameters
- \$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\_INP\_MAP: Configuration for system bits in Input on fieldbuses
- \$SYS\_OUT\_MAP: Configuration for system bits in Output on fieldbuses
- \$SYS\_PARAMS: Robot system identifier
- \$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
- \$TOL\_ABST: 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\_FRICTION: Tool Friction
- \$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
- \$TUNE: Internal tuning parameters
- \$TURN\_CARE: Turn care
- \$TX\_RATE: Transmission rate
- \$UFRAME: User frame of an arm
- \$USER\_ADDR: Address of user-defined variables
- \$USER\_BIT: User-defined bit memory
- \$USER\_BYTE: User-defined byte memory
- \$USER\_LEN: Length of User-defined variables
- \$USER\_LONG: User-defined long word memory
- \$USER\_WORD: User-defined word memory
- \$VERSION: Software version
- \$VP2\_SCRN\_ID: Executing program VP2 Screen Identifier
- \$VP2\_TUNE: Visual PDL2 tuning parameters
- \$VP2\_TOUT: Timeout value for asynchronous VP2 requests
- \$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: PLC WORD data
- \$WRITE\_TOUT: Timeout on a WRITE
- \$WV\_AMP\_PER: Weave amplitude percentage
- \$WV\_CNTR\_DLW: Weave center dwell
- \$WV\_LEFT\_AMP: Weave left amplitude
- \$WV\_LEFT\_DLW: Weave left dwell
- \$WV\_LENGTH\_WAVE: Wave length
- \$WV\_ONE\_CYCLE: Weave one cycle
- \$WV\_PLANE: Weave plane angle
- \$WV\_RIGHT\_AMP: Weave right amplitude
- \$WV\_RIGHT\_DLW: 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 \$AIN: Analog input

<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>S/W Version:</i>	1.00
<i>Description:</i>	It represents the analog input points. For further information see also <a href="#">par. 5.2.4 \$AIN and \$AOUT on page 5-5</a> and <b>Control Unit Use Manual</b> - Chapters IO_INST Program and IO_TOOL Program.

## 12.11 \$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>S/W Version:</i>	1.00
<i>Description:</i>	It represents the analog output points. For further information see also <a href="#">par. 5.2.4 \$AIN and \$AOUT on page 5-5</a> and <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.12 \$APPL\_ID: Application Identifier

<i>Memory category</i>	field of board_data
<i>Load category:</i>	retentive
<i>Data type:</i>	string
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	It's a field of board_data that contains information about an application

## 12.13 \$APPL\_NAME: Application Identifiers

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category</i> - environment
<i>Data type:</i>	array of string of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.20
<i>Description:</i>	Each element of this array is a string that contains the name of an application that is running on the Controller

## 12.14 \$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>S/W Version:</i>	2.20
<i>Description:</i>	This variable contains the definitions of the application features currently enabled on the Controller

## 12.15 \$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>	WITH MOVE; MOVE ALONG
<i>Limits</i>	1..100
<i>S/W Version:</i>	1.00
<i>Description:</i>	It represents the acceleration override percentage for motions issued to a particular 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

## 12.16 \$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>S/W Version:</i>	1.00
<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 arndata PROG_ARM=1 VAR init_tool : POSITION BEGIN \$ARM_DATA[2].TOOL := init_tool -- arm 2 \$TOOL := init_tool -- arm 1 END arndata

## 12.17 \$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>S/W Version:</i>	1.00

*Description:* 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

## 12.18 \$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>S/W Version:</i>	1.00
<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

## 12.19 \$ARM\_LINKED: Enable/disable arm coupling

<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>	WITH MOVE
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	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.

## 12.20 \$ARM\_OVR: Arm override

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - overrides
<i>Data type:</i>	integer

*Attributes:* none  
*Limits* 1..100  
*S/W Version:* 1.00  
*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.

## 12.21 \$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  
*S/W Version:* 1.00  
*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

## 12.22 \$ARM\_SIMU: Arm simulate flag

*Memory category* field of crnt\_data  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.23 \$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  
**S/W Version:** 2.4x  
**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
    
```

## 12.24 \$ARM\_SPD\_OVR: Arm speed override

**Memory category** field of arm\_data  
**Load category:** arm. *Minor category* - overrides  
**Data type:** integer  
**Attributes:** WITH MOVE; MOVE ALONG  
**Limits** 1..100  
**S/W Version:** 1.00  
**Description:** It represents the speed override percentage for motions issued to a particular arm. There is one value per arm. Acceleration and deceleration are not effected by changes in its value. Changes in \$ARM\_SPD\_OVR take effect on the next motion and for the entire motion.

## 12.25 \$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  
**S/W Version:** 1.00  
**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.

## 12.26 \$ARM\_VEL: Arm velocity

<i>Memory category:</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	real
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>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 5 ticks in case of joint movements.</p> <pre>PROGRAM vel NOHOLD BEGIN CYCLE   WRITE (\$CRNT_DATA[1].ARM_VEL, NL)   DELAY 500 END vel</pre>

## 12.27 \$AUX\_BASE: Auxiliary base for a positioner of an arm

<i>Memory category:</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - auxiliary</i>
<i>Data type:</i>	array of position of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>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.</p>

## 12.28 \$AUX\_KEY: TP4i/WiTP AUX-A and AUX-B keys mapping

<i>Memory category:</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category - auxiliary</i>
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	7..10
<i>S/W Version:</i>	1.00

*Description:* This variable contains the indication of the auxiliary axis that is associated to the corresponding key on the Teach Pendant. On TP4i/WiTP, element 1 is related to the AUX-A key and element 2 is related to AUX-B key.

## 12.29 \$AUX\_MASK: Auxiliary arm mask

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category - auxiliary*  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* Note that 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.

## 12.30 \$AUX\_OFST: Auxiliary axes offsets

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category - auxiliary*  
*Data type:* array of real of one dimension  
*Attributes:* WITH MOVE; MOVE ALONG  
*Limits*  
*S/W Version:* 1.00  
*Description:* 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 in fact, \$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 DISPLAY ARM STATUS takes into account these offsets.

## 12.31 \$AUX\_SIK\_DRVON\_ENBL: Auxiliary axes provided of SIK

*Memory category* field of crnt\_data

*Load category:* not saved. *Minor category - auxiliary*  
*Data type:* integer  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 2.20  
*Description:* Each bit in the INTEGER represents whether the corresponding auxiliary axis, supplied with a Safety Interlock Kit module, is enabled to DRIVE ON

## 12.32 \$AUX\_SIK\_MASK: Auxiliary axes provided of SIK

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category - auxiliary*  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 2.20  
*Description:* Each bit in the INTEGER represents whether the corresponding auxiliary axis is supplied with a Safety Interlock Kit module or not

## 12.33 \$AUX\_TYPE: Positioner type

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category - configuration*  
*Data type:* array of integer of two dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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

## 12.34 \$AX\_CNVRSN: Axes conversion

*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  
*S/W Version:* 1.00  
*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

```

## 12.35 \$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  
*S/W Version:* 1.00  
*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

```

## 12.36 \$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  
*S/W Version*: 1.00  
*Description*: it represents the axes lengths of the arm measured in millimeters.

## 12.37 \$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*: read-only  
*Limits*: none  
*S/W Version*: 1.00  
*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.

## 12.38 \$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  
*S/W Version*: 1.00  
*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.

## 12.39 \$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:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.40 \$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  
*S/W Version:* 1.00  
*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.

## 12.41 \$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:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.42 \$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

*S/W Version:* 2.00

*Description:* Element 1 contains the BSP version. Element 2 contains the date of the BSP generation.

## 12.43 \$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

*S/W Version:* 1.00

*Description:* It is an array of 8 elements with each element representing the definition of a backup save set.

These save sets are used by the FilerUtilityBackup command, when the option /Saveset 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.

As a default element [1] contains the value "All|UD:\\*.\*\s".

## 12.44 \$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

*S/W Version:* 1.00

*Description:* It represents the location and orientation of the base of the robot relative to world frame of reference.

## 12.45 \$BIT: PLC BIT data

*Memory category* port

*Load category:* not saved

*Data type:* array of boolean of one dimension

*Attributes:* pulse usable

*Limits* none

*S/W Version:* 1.00

*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 5-16](#) and **Control Unit Use** Manual - Chapters IO\_INST Program and IO\_TOOL Program.

## 12.46 \$BOARD\_DATA: Board data

*Memory category* dynamic  
*Load category:* not saved  
*Data type:* array of board\_data of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* BOARD\_DATA is a record system variable that contains detailed information about each board mounted in the rack. The first board is always an SMP board and after that N MCP boards (if present). The data per record is specific per board.

## 12.47 \$BOOTLINES: Bootline read-only

*Memory category* field of board\_data  
*Load category:* retentive  
*Data type:* array of string of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.48 \$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  
*S/W Version:* 1.00  
*Description:* Boolean registers are “” variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position and Xtdpos there are saved and non-saved registers

## 12.49 \$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  
*S/W Version*: 1.00  
*Description*: Boolean registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position and Xtndpos there are saved and non-saved registers

## 12.50 \$B\_ALONG\_1D: Internal arm data

*Memory category*: field of board\_data  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 1.00  
*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.

## 12.51 \$B\_ALONG\_1D\_NS: Internal arm data

*Memory category*: field of board\_data  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 1.00  
*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.

## 12.52 \$B\_NVRAM: NVRAM data of the board

*Memory category*: field of board\_data  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: There are certain fields of \$BOARD\_DATA which are also saved into NVRAM and 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

## 12.53 \$C4GOPEN\_JNT\_MASK: C4G Open Joint arm mask

*Memory category*: field of arm\_data  
*Load category*: arm. *Minor category* - configuration  
*Data type*: integer  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 3.1  
*Description*: Each bit in the INTEGER represents whether the corresponding axis for that arm is enabled for the C4GOPEN modality.

## 12.54 \$C4GOPEN\_MODE: C4G Open modality

*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  
*S/W Version*: 3.1  
*Description*: Each array element contains an integer which identifies the C4GOpen modality foreseen for that axis.

## 12.55 \$C4G\_RULES: C4G Save & Load rules

*Memory category* static  
*Load category:* controller. *Minor category* - environment  
*Data type:* arrays of string of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.20  
*Description:* It is an array of 8 elements with each element representing the definition of a C4G save/load. Each element can be used defining what gets saved/loaded with a C4G file when using Configure Save or Configure Load. The string format 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\*

## 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* none  
*S/W Version:* 1.00  
*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  
*S/W Version:* 1.00

*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

*S/W Version:* 1.00

*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

*S/W Version:* 1.00

*Description:* It contains the cartesian position of the enabled positioner for the cooperative motion.

## 12.60 \$CIO\_AIN: Configuration for AIN

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Analog Inputs. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.61 \$CIO\_AOUT: Configuration for AOUT

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Analog Outputs. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.62 \$CIO\_CAN: Configuration for Can Bus

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of the user modules on Can Bus. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.
[1]	Reserved
[2]	Module address

- [3] Device Type
- [4] Vendor Id
- [5] Product Code
- [6] Mask of the possible inputs
- [7] Reserved
- [8] mask of the possible outputs
- [9] Reserved
- [10] Voltage ranges. Used only for the analog modules
- [11] Reserved
- [12] Reserved
- [13] Reserved

## 12.63 \$CIO\_CROSS: Configuration for I/O cross copying

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category - configuration</i>
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	<p>This table contains information about the I/O copy feature. It is possible to copy I/O points each other and this variable contains the description of the association between input and output ports.</p> <p>For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.</p>

## 12.64 \$CIO\_DIN: Configuration for DIN

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category - configuration</i>
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	<p>This table contains information about the configuration of Digital Inputs. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.</p>

## 12.65 \$CIO\_DOUT: Configuration for DOUT

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Digital Outputs. For further information see also <b>Control Unit Use</b> Manual - Chap. IO_INST Program-I/O Configuration.

## 12.66 \$CIO\_FMI: Configuration for \$FMI

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	2.20
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Flexible Multiple Input Points. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.67 \$CIO\_FMO: Configuration for \$FMO

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	2.20
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Flexible Multiple Output Points. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.68 \$CIO\_GIN: Configuration for GIN

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category - configuration</i>
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Digital Input Groups. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.69 \$CIO\_GOUT: Configuration for GOUT

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category - configuration</i>
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Digital Output Groups. For further information see also <b>Control Unit Use</b> Manual - Chapters IO_INST Program and IO_TOOL Program.

## 12.70 \$CIO\_IN\_APP: Configuration for IN

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category - application</i>
<i>Data type:</i>	of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Digital Applications Inputs.

## 12.71 \$CIO\_OUT\_APP: Configuration for OUT

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - application
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of Digital Applications Outputs.

## 12.72 \$CIO\_SDIN: Configuration for the system digital inputs

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex
<i>Description:</i>	This table contains information about the configuration of the system digital inputs: [1] - description of the fieldbus type [2] - Index [3] - details about where this inputs is physically coming from

## 12.73 \$CIO\_SDOOUT: Configuration for the system digital outputs

<i>Memory category</i>	static
<i>Load category:</i>	input/output. <i>Minor category</i> - configuration
<i>Data type:</i>	array of integer of two dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Unparsed</i>	Hex

*Description:* This table contains information about the configuration of the system digital outputs:  
 [1] - description of the fieldbus type  
 [2] - Index  
 [3] - details about where this outputs is physically going to

## 12.74 \$CIO\_SYS\_CAN: Configuration of the system modules on Can Bus

*Memory category* static  
*Load category:* input/output. *Minor category* - configuration  
*Data type:* array of integer of two dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Unparsed* Hex  
*Description:* This table contains information about the configuration of the system modules on Can Bus. For further information see also **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.75 \$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  
*S/W Version:* 1.00  
*Description:* Represents a flag to determine whether a Cartesian trajectory with a different starting and ending configuration should be executed. If the flag is TRUE and the initial and final configurations are different, the motion is not executed. If the flag is FALSE, the final configuration will be the same as the initial configuration without any messages. The default value is TRUE.

## 12.76 \$CNTRL\_CNFG: Controller configuration mode

*Memory category* static  
*Load category:* controller. *Minor category* - environment  
*Data type:* integer

<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>Different bits of this predefined variable are used for setting the configuration and startup of the controller:</p> <ul style="list-style-type: none"> <li>– Bit 1: If set, the pop-up window containing informations on the system configuration is not displayed at system restart.</li> <li>– Bit 2: reserved</li> <li>– 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: <ul style="list-style-type: none"> <li>• a: if in PROGR state from an active program.</li> <li>• b: if in PROGR state executing the statement from the WINC4G program running on the PC when the Teach Pendant is recognised to be out of cabinet.</li> <li>• c: if in PROGR state under MEMORY DEBUG or PROGRAM EDIT from the WINC4G program running on the PC when the Teach Pendant is recognised to be out of cabinet.</li> <li>• d: if the state selector was turned out of the T1 position when the Teach Pendant is recognised to be out of cabinet.</li> </ul> </li> <li>– Bit 4: reserved</li> <li>– 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)</li> <li>– Bit 6: reserved</li> <li>– Bit 7: If set, the communication protocol for WINC4G program is automatically mounted on COMP: port after a restart.</li> <li>– Bit 8: reserved</li> <li>– Bit 9: If set, the EZ emulator does not automatically start when EZ is activated.</li> <li>– Bit 10: If set, EZ does not wait for the UTILITY APPLICATION command or the EZ key to be pressed.</li> <li>– Bit 11: If set, EZ does not load in memory the programs that are referenced in the main program.</li> <li>– Bit 12: If set, EZ does not make any checks about the applications during the installation phase</li> <li>– Bit 13: Set to 1 for handling MOVE WITH \$PAR.</li> <li>– Bit 14-15: reserved to EZ</li> <li>– Bit 16-22</li> <li>– Bit 23: set to 1 to enable Autosave in IDE</li> <li>– Bit 24: Set to 1 if the system is configured to handle the Customized Nodal Move</li> <li>– Bit 25: Set to 1 if the UNICODE characters coding is handled by the system</li> <li>– Bit 26-32: reserved</li> </ul>

## 12.77 \$CNTRL\_INIT: Controller initialization mode

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category</i> - environment
<i>Data type:</i>	integer
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00

- 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, PROGR and AUTO-T 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, CAC and CAT 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.

## 12.78 \$CNTRL\_OPTIONS: Controller Options

- Memory category* static
- Load category:* controller
- Data type:* array of integer of one dimension
- Attributes:* read-only
- Limits* none
- S/W Version:* 1.20
- Description:* Each bit of this variable correspond 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 Absolute accuracy algorithm
  - Bit 7 for Collision detection algorithm
  - Bit 8 for Automatic payload identification algorithm
  - Bit 9 for Complying arm technology (joint or Cartesian Soft Servo)
  - Bit 10 for Software PLC
  - Bit 11 for Ethernet/IP Master
  - Bit 12 for Ethernet/IP Slave
  - Bit 13 for reading/writing on socket PDL2
  - Bit 14 for USB Belkin
  - Bit 15 for Modem Remote Connection
  - Bit 16 for Arm Motion with Speed Control

- Bit 17 for Modbus TCP Slave
- Bit 18 for Modbus TCP Master
- Bit 19 for SmartMove4
- Bit 20 for Path Governor
- Bit 22 for VP2Frames
- Bit 23 reserved
- Bit 24 for Profinet I/O master
- Bit 25 for Multibus
- Bit 26 for C4G Open

## 12.79 \$CNTRL\_TZ: Controller Time Zone

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category</i> - environment
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	2.20
<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>

## 12.80 \$COLL\_EFFECT: Collision Effect on the arm status

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	0..2
<i>S/W Version:</i>	2.2
<i>Description:</i>	<p>It is possible to define the effect of the Collision Alarm (<b>62513 - Collision detected</b>) on the Arm. This predefined variable can assume the following values:</p> <ul style="list-style-type: none"> <li>– 0: a DRIVE OFF is issued. This is the default value.</li> <li>– 1: a HOLD is generated</li> <li>– 2: the collision causes an event which can be tested using WHEN EVENT 197.</li> </ul>

## 12.81 \$COLL\_ENBL: Collision enabling flag

<i>Memory category</i>	field of crnt_data
------------------------	--------------------

*Load category:* not saved  
*Data type:* boolean  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* When set to TRUE, this flag enables the "Collision detection" feature and has an immediate effect on any movement.

## 12.82 \$COLL\_SOFT\_PER: Collision compliance percentage

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - collision  
*Data type:* array of integer of two dimension  
*Attributes:* WITH MOVE  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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.  
 The first dimension is the one of the possible values of \$COLL\_TYPE. The second one is the axis number.

## 12.83 \$COLL\_TYPE: Type of collision

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - collision  
*Data type:* integer  
*Attributes:* WITH MOVE; MOVE ALONG  
*Limits* COLL\_LOW..COLL\_USER2  
*S/W Version:* 1.00  
*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 and of \$COLL\_SOFT\_PER 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 informational 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

## 12.84 \$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

*S/W Version:* 1.00

*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
    
```

## 12.85 \$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>S/W Version:</i>	1.00
<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.)

## 12.86 \$CONV\_ACC\_LIM: Conveyor acceleration limit

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - conveyor
<i>Data type:</i>	array of real of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	It represents the maximum acceleration/deceleration used by the arm to track the conveyor

## 12.87 \$CONV\_BASE: Conveyor base frame

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - conveyor
<i>Data type:</i>	array of position of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.0
<i>Description:</i>	<p>It indicates the position of the conveyor base with respect to the robot world frame. It must be measured when the conveyor-truck is at the zero position or in a known position. The X axis of the resulting frame of reference (conveyor base frame) must be aligned with the conveyor direction; its X-Y plane coincides with the plane of the conveyor-truck and its origin is located in the point where the passage of the truck is detected by a sensor. If the conveyor is circular the Y axis must be oriented towards the center of the rotation. Given three points on the conveyor (origin, x, xy) the \$CONV_BASE can be computed by means of the POS_FRAME built-in</p>

## 12.88 \$CONV\_CNGF: Conveyor tracking configuration

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - conveyor
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.0\0
<i>Description:</i>	<p>It is a bit mask that configures the conveyor-belt. The first bit specifies the type of conveyor tracking: FALSE if cartesian and TRUE if rail tracking. The second bit specifies the direction of the rail: TRUE if the positive direction of the conveyor is the positive direction of the rail. The third bit specifies if the cartesian tracking is performed on a linear (FALSE) or circular belt. This configuration is used either for conveyor tracking and for conveyor reading. This variable is initialized by the PDL2 configuration tool</p>

## 12.89 \$CONV\_DIST: Conveyor shift in micron (mm/1000)

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none

*S/W Version:* 1.00  
*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. It can be read when the conveyor is active either in tracking mode or in reading mode (\$CONV\_TYPE different from zero). As it is an INTEGER variable, \$CRNT\_DATA[n].CONV\_DIST is mainly useful inside the CONDITIONS

## 12.90 \$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  
*S/W Version:* 1.00  
*Description:* The \$CRNT\_DATA[n].CONV\_SHIFT represents the shift of the conveyor truck frame with respect to the zero position (conveyor base frame). It is expressed in millimeters in case of linear conveyor and in degrees in case of circular conveyor. It can be read when the conveyor is active either in tracking mode or in reading mode.

## 12.91 \$CONV\_SPD: Conveyor speed

*Memory category* field of crnt\_data  
*Load category:* not saved  
*Data type:* array of real of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* The \$CRNT\_DATA[n].CONV\_SPD represents the conveyor speed in meters per second. It can be used to monitor the process. It can be read when the conveyor is active either in tracking mode or in reading mode.

## 12.92 \$CONV\_SPD\_LIM: Conveyor speed limit

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - conveyor  
*Data type:* array of real of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00

*Description:* It represents the maximum speed used by the arm to track the conveyor

## 12.93 \$CONV\_TBL: Conveyor tracking table data

*Memory category* dynamic

*Load category:* not saved

*Data type:* array of conv\_tbl of one dimension

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*Description:* \$CONV\_TBL is an array of conveyor schedules with each schedule containing the \$CT\_xxx fields

## 12.94 \$CONV\_TYPE: Element in the Conveyor Table

*Memory category* field of arm data

*Load category:* arm. *Minor category* - conveyor

*Data type:* integer

*Attributes:* none

*Limits* CONV\_READ..CONV\_1READ\_2ON

*S/W Version:* 1.00

*Description:* Represents the Conveyor Table (\$CONV\_TBL) element to be used in the next motions. Set this variable to CONV\_OFF to disable both the conveyor tracking modalities; set it to the values CONV\_1ON or CONV\_2ON to enable the tracking on the first or second conveyor belt; set to the values CONV\_1READ or CONV\_2READ to enable the reading on the first or second conveyor belt; set it to CONV\_1ON\_2READ, to enable the tracking on the first and to enable the reading on the second; set it to the values CONV\_1READ\_2ON to enable the tracking on the second and the reading on the first.

## 12.95 \$CONV\_WIN: Conveyor Windows

*Memory category* field\_of\_arm\_data

*Load category:* arm. *Minor category* - conveyor

*Data type:* array of real of one dimension

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*Description:* It defines the tracking window that is a region where conveyor tracking can be performed. \$CONV\_WIN[1] contains the distance of the inbound boundary from the conveyor base frame and \$CONV\_WIN[2] contains the distance of the outbound boundary. If the robot attempts to move a position when the part has not yet come within the window, the Controller Unit waits until the part enters the window. If the robot attempts to move to a position when the part has moved out of the window then the Controller Unit generates an error.

## 12.96 \$CONV\_ZERO: Conveyor Position Transducer Zero

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - conveyor  
*Data type:* array of real of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable has the meaning of indicating that the truck is in the zero position and in this case it should be set to zero (\$CONV\_ZERO[conv\_num] := 0.0) by the application package which monitors this event. The system guarantees that this variable always contains the shift of the truck, expressed in motor turns, in respect to the latest setting to zero

## 12.97 \$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  
*S/W Version:* 1.00  
*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

## 12.98 \$CT\_JNT\_MASK: Conveyor Joint mask

*Memory category* field of conv\_tbl  
*Load category:* controller. *Minor category* - conveyor  
*Data type:* integer  
*Attributes:* none

<i>Limits</i>	0..1023
<i>S/W Version:</i>	1.00
<i>Description:</i>	It is used in conveyor tracking applications where an external resolver is connected to the conveyor motor. The variable is a bit mask specifying the physical axis where the resolver is connected

## 12.99 \$CT\_RADIUS: Conveyor radius in mm

<i>Memory category</i>	field of conv_tbl
<i>Load category:</i>	controller. <i>Minor category - conveyor</i>
<i>Data type:</i>	real
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	The \$CONV_TBL[n].CT_RADIUS it is useful for circular cartesian tracking and represents the radius of the conveyor with respect to the zero position (conveyor base frame origin). It is expressed in millimeters

## 12.100 \$CT\_RES: Conveyor position in motor turns

<i>Memory category</i>	field of conv_tbl
<i>Load category:</i>	controller. <i>Minor category - conveyor</i>
<i>Data type:</i>	real
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	The \$CONV_TBL[n].CT_RES variable represents the position of the conveyor, expressed in motor turns of the conveyor motor. It is used to reset the conveyor offset copying the value in the \$CONV_ZERO variable

## 12.101 \$CT\_SCC: Conveyor SCC

<i>Memory category</i>	field of conv_tbl
<i>Load category:</i>	controller. <i>Minor category - conveyor</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	0..2
<i>S/W Version:</i>	1.00

*Description:* It is used in conveyor tracking applications where an external resolver is connected to the conveyor motor. The variable indicates the number of the servo control card to which the resolver is connected

## 12.102 \$CT\_TX\_RATE: Transmission rate

*Memory category* field of conv\_tbl  
*Load category:* controller. *Minor category* - conveyor  
*Data type:* real  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* The conveyor scaling factor. It is expressed in [mm/motor\_turn] for the linear cartesian tracking and rail tracking; it is expressed in [motor\_turns/conveyor\_turn] for circular tracking

## 12.103 \$CUSTOM\_ARM\_ID: Identifier for the arm

*Memory category* static  
*Load category:* controller. *Minor category* - environment  
*Data type:* array of string of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.20  
*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

## 12.104 \$CUSTOM\_CNTRL\_ID: Identifier for the Controller

*Memory category* static  
*Load category:* controller. *Minor category* - environment  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.20  
*Description:* This string element contains the identification name assigned by the user to the Controller

## 12.105 \$CYCLE: Program cycle count

<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>S/W Version:</i>	1.00
<i>Description:</i>	<p>It represents a counter for the number of cycles the program has executed. If the CYCLE option is not on the <b>BEGIN Statement</b>, the initial value is 0. Otherwise, it is 1. It is incremented each time a program <b>END Statement</b> or <b>EXIT CYCLE Statement</b> is executed. This variable can be useful for initialization.</p>

```

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

```

## 12.106 \$C\_ALONG\_1D: Internal current arm data

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>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</p>

## 12.107 \$C\_AREAL\_1D: Internal current arm data

<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>	privileged read-write

*Limits* none  
*S/W Version:* 1.00  
*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

## 12.108 \$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  
*S/W Version:* 1.00  
*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

## 12.109 \$DEPEND\_DIRS: Dependancy path

*Memory category* static  
*Load category:* controller. *Minor category - environment*  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.110 \$DFT\_ARM: Default arm

*Memory category* static  
*Load category:* controller. *Minor category - environment*  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* none

*S/W Version:* 1.00

*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

## 12.111 \$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

*S/W Version:* 1.00

*Description:* It represents the various default devices used by the controller. It is an array of 6 elements having the following meaning:

- [1]: Default PDL2 device ('TP:')
- [2]: Default system device ('UD:')
- [3]: Default backup device ('XD:')
- [4]: Default device for temporary files ('TD:')
- [5]: Default communication device ('COM1:')
- [6]: Default device for PDL2 ('COMP:')
- [7]: Default device for installation (file Utility Install) ('COMP:')
- [8]: Default device for COMP ('NET0:')

## 12.112 \$DFT\_LUN: Default LUN number

*Memory category* static

*Load category:* not saved

*Data type:* integer

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*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 Winc4g 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

## 12.113 \$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  
*S/W Version:* 1.00  
*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 Winc4g)  
[3..4] reserved

## 12.114 \$DIN: Digital input

*Memory category* port  
*Load category:* not saved  
*Data type:* array of boolean of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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 5-4](#) and **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.115 \$DNS\_DOMAIN: DNS Domain

*Memory category* field of board\_data  
*Load category:* controller. *Minor category - shared*  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This is the name of the DNS Domain from where IP address will be located

## 12.116 \$DNS\_ORDER: DNS Order

*Memory category*: field of board\_data  
*Load category*: controller.    *Minor category* - shared  
*Data type*: integer  
*Attributes*: none  
*Limits*: 1..3  
*S/W Version*: 1.00  
*Description*: Order in which addresses are searched:  
 1: Local host table  
 2: DNS server first  
 3: DNS server only

## 12.117 \$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  
*S/W Version*: 1.00  
*Description*: An array that can contain up to 3 IP addresses for the location of the DNS server

## 12.118 \$DOUT: Digital output

*Memory category*: port  
*Load category*: not saved  
*Data type*: array of boolean of one dimension  
*Attributes*: pulse usable  
*Limits*: none  
*S/W Version*: 1.00  
*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 5-4 Control Unit Use Manual - Chapters IO\\_INST Program and IO\\_TOOL Program.](#)

## 12.119 \$DSA\_DATA: DSA data

*Memory category*: dynamic  
*Load category*: not saved  
*Data type*: array of dsa\_data of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: \$DSA\_DATA is an array of predefined records with one element for each Motion Control Process. The fields of each \$DSA\_DATA element represent the Digital Signal Amplifier (DSA) data. Each element is an electro-mechanical peculiarity of the machine and the servo-control parameters

## 12.120 \$DV\_STS: the status of DV4\_CNTRL call

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..0  
*S/W Version*: 1.00  
*Description*: This variable contains the status of the last DV4\_CNTRL operation

## 12.121 \$DV\_TOUT: Timeout for asynchronous DV4\_CNTRL calls

*Memory category*: static  
*Load category*: controller.     *Minor category - environment*  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..MAXINT  
*S/W Version*: 1.00  
*Description*: This variable contains the timeout value for asynchronous DV4\_CNTRL calls with a value of zero meaning an indefinite wait

## 12.122 \$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  
*S/W Version*: 1.00  
*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

## 12.123 \$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*: 0..100  
*S/W Version*: 2.20  
*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.

## 12.124 \$DYN\_FILTER: Dynamic Filter

*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  
*S/W Version*: 1.00  
*Description*: This array represents the coefficient of the position, velocity and acceleration filters used in the new dynamic model algorithm

## 12.125 \$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  
*S/W Version*: 2.00  
*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 in the DSA 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.

## 12.126 \$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  
*S/W Version*: 2.00  
*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.

## 12.127 \$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  
*S/W Version*: 1.00  
*Description*: This is a set of data for the new dynamic model algorithm that represents the robot identification parameters without payload

## 12.128 \$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  
*S/W Version:* 1.00  
*Description:* This is a set of data for the new dynamic model algorithm that represents the wrist identification parameters without load

## 12.129 \$DYN\_WRISTQS: Dynamic Theta carico

*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  
*S/W Version:* 1.00  
*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

## 12.130 \$D\_ALONG\_1D: Internal DSA data

*Memory category* field of dsa\_data  
*Load category:* DSA  
*Data type:* array of integer of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* There are certain fields of \$DSA\_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 unspecified

## 12.131 \$D\_AREAL\_1D: Internal DSA data

*Memory category* field of dsa\_data

*Load category:* DSA  
*Data type:* array of real of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* There are certain fields of \$DSA\_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 unspecified

## 12.132 \$D\_AXES: Internal DSA data

*Memory category* field of dsa\_data  
*Load category:* DSA  
*Data type:* array of integer of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* There are certain fields of \$DSA\_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

## 12.133 \$D\_CTRL: Internal DSA data

*Memory category* field of dsa\_data  
*Load category:* DSA  
*Data type:* array of real of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* There are certain fields of \$DSA\_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

## 12.134 \$D\_HDIN\_SUSP: DSA\_DATA field for HDIN suspend

*Memory category* field of dsa\_data  
*Load category:* DSA  
*Data type:* integer

<i>Attributes:</i>	privileged read-write
<i>Limits</i>	0, 1
<i>S/W Version:</i>	2.4x
<i>Description:</i>	<p>there is one of this variable for each DSA board.</p> <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 function.</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 is called, the \$HDIN_SUSP is automatically set to 0. It is therefore suggested to use the \$HDIN_SUSP only after the call to HDIN_SET</p>

## 12.135 \$D\_MTR: Internal DSA data

<i>Memory category</i>	field of dsa_data
<i>Load category:</i>	DSA
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	privileged read-write
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>There are certain fields of \$DSA_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</p>

## 12.136 \$EMAIL\_INT: Email integer configuration

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category</i> - unique
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	2.20
<i>Description:</i>	<p>[1]: flags</p> <ul style="list-style-type: none"> <li>• 0x01: Feature is enabled</li> <li>• 0x04: use APOP authentication</li> <li>• 0x10: Use of memory or file system for incoming mail</li> <li>• 0x020: Do not remove the directory where attachments are stored upon system startup</li> </ul> <p>[2]: Maximum size in bytes for incoming messages (if 0, the default is 300K, that is 300 * 1024 bytes)</p> <p>[3]: Polling interval for POP3 server (if 0, the default and minimum used value is 60000 ms)</p> <p>[4]: Size to reserve to the body of the email (if 0 the default value is 5K, that is 5 * 1024 bytes)</p> <p>[5]: timeout (if 0 the default value is 10000 ms)</p>

## 12.137 \$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  
*S/W Version*: 2.20  
*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

## 12.138 \$ERROR: Last PDL2 Program Error

*Memory category*: program stack  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..0  
*S/W Version*: 1.00  
*Description*: This variable contains the last error (if any) that occurred in the program

## 12.139 \$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  
*S/W Version*: 1.00  
*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

## 12.140 \$FBP\_TBL: Field Bus Table data

*Memory category* dynamic  
*Load category:* not saved  
*Data type:* array of fbp\_tbl of one dimension  
*Attributes:* none  
*Limits* 1..4.  
 Meaningful values:  
 1 and 2: FBP boards plugged in the electronic rack  
 3 : reserved for fieldbus on Ethernet  
 4 : reserved for fieldbus on CAN  
*S/W Version:* 1.00  
*Description:* Each fieldbus processor board (FBP) has its own element in this table. Those tables are used by IO\_TOOL program in order to configure fieldbuses..

## 12.141 \$FB\_ADDR: Field Bus ADDR

*Memory category* field of fbp\_tbl  
*Load category:* input/output. *Minor category - fieldbus*  
*Data type:* integer  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This field contains the physical address of the FBP dual port memory

## 12.142 \$FB\_CNFG: Controller fieldbuses configuration mode

*Memory category* static  
*Load category:* not saved  
*Data type:* array of integer of two dimension  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* Only 3 elements of this variable are currently used. Element 1 is reserved; element 2 identifies the Profibus DP interface, element 3 identifies the Interbus interface; element 4 identifies the Device Net interface.  
 Each element is a bit mask where each bit represents the different fieldbus options present in the system:  
 Bit 1..6: reserved

Bit 7: Fieldbus primary interface configured correctly

Bit 8..22: reserved

Bit 23: Slave interface configured

Bit 24: Master interface configured

Bit 25..29: reserved

Bit 30: Fieldbus processor board plugged in

Bit 31..32: reserved

For further information see also **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.143 \$FB\_INIT: Controller fieldbuses initialization mode

*Memory category*: static

*Load category*: input/output    *Minor category* - fieldbus

*Data type*: array of integer of two dimension

*Attributes*: privileged read-write

*Limits*: none

*S/W Version*: 1.00

*Description*: Only 3 elements of this variable are currently used. Element 1 is reserved; element 2 identifies the Profibus DP interface, element 3 identifies the Interbus interface; element 4 identifies the Device Net interface.

Each element is a bit mask where each bit represents how different aspects of the Controller behave upon initialization:

- Bit 1..7: reserved
- Bit 8,9,10: these three bits configure how the system \$WORD elements are handled by the C4G Controller. The following table shows the current usage:

System \$WORD		\$CNTRL_INIT
Usage	PDL2 access	
----- UNINIT -----		011
---- Other \$WORDS ----		010
User	\$WORD	111
Sys	\$SDIN/SDOUT	000
----- Reserved -----		001
----- Reserved -----		100
----- Reserved -----		101
----- Reserved -----		110

- Bit 11,15: reserved
- Bit 16, 32: reserved

Note that for making operative any setting of \$FB\_INIT, it is needed to restart the Controller.

For further information see also **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.144 \$FB\_MA\_INIT: Field bus master init

*Memory category*: field of fbp\_tbl  
*Load category*: input/output. *Minor category* - fieldbus  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: the variable contains initialization and setup information for the MASTER of the field bus communication.

It must be properly initialized before using the board in the Controller.

The following table shows the meaning of each element:

[1]: configuration options flag

- Bit 1: if set to 1, it indicates that the Master is enabled. Otherwise the Controller assumes that the Master is not present. For making changes on this bit to operate, it is needed to restart the Controller.

[2]: Baud Rate

For each type of fieldbus network a specific value must be set:

- for the Profibus - 1: 9600 baud; 2: 19200 baud; 3: 31250 baud; 4: 44450 baud; 5: 93750 baud; 6: 187.5 Kbaud; 8: 500 Kbaud; 9: reserved; 10: 1,5 Mbaud; 11: 3 Mbaud; 12: 6 Mbaud; 13: 12 Mbaud
- for the Device Net - 1: 125 Kbaud; 2: 250 Kbaud; 3: 500 Kbaud
- for the Interbus: the baud rate is automatically set by the system

[3]: address of the Master on the fieldbus network

[4]: reserved

[5]: start index of \$WORD for commands to the master. The node index used by these commands must be specified 3 \$WORDs after this one. Possible commands to the Master:

- network start (bit 1 set to 1)
- network stop (bit 2 set to 1)
- node reset (bit 3 set to 1)
- node start (bit 4 set to 1)
- node stop (bit 5 set to 1)

[6]: index of \$WORD for the status of the command. When a command has been executed, the corresponding bit is set to 1. The bit is cleared when the command is cleared

[7]: connection timeout in msec after a start command on a profibus node.  
0 means 5 seconds

[8..20]: reserved

## 12.145 \$FB\_MA\_SLVS: Field bus master slaves init

*Memory category*: field of fbp\_tbl  
*Load category*: input/output. *Minor category* - fieldbus  
*Data type*: array of integer of two dimension  
*Attributes*: none

<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>the variable contains initialization and setup information for the slaves on a fieldbus when the C4G is a master.</p> <p>The following table shows the meaning of each element:</p> <ul style="list-style-type: none"> <li>[1]: configuration flags                     <ul style="list-style-type: none"> <li>– Bit 1..3: reserved</li> <li>– Bit 4: if 0 there is the digital copy in the \$DIN, \$DOUT. If 1 the copy is on \$WORD.</li> <li>– Bit 5: if 0 the node is active as default; if 1 the node is active</li> <li>– Bit 6: for Profibus only. If 0 the node inputs and outputs are handled as single bytes; if 1 they are treated as words (2 bytes)</li> <li>– Bit 7: for Profibus only. If 0 the word consistency is disabled; if 1 it is enabled</li> <li>– Bit 8: for profibus only. If 0 the blocks consistency is disabled if 1 it is enabled</li> <li>– Bit 9..32: reserved</li> </ul> </li> <li>[2]: number of input bytes</li> <li>[3]: Network address</li> <li>[4]: node identifier for Profibus and Interbus; Vendor ID for DeviceNet</li> <li>[5]: product code for DeviceNet only</li> <li>[6..7]: reserved</li> <li>[8]: start index of user \$WORDs</li> <li>[9]: number of output bytes</li> <li>[10]: for Profibus only. First set of configuration parameters. The least significant byte indicates the number of parameters (possible value from 0 to 7). The remaining bytes hold the parameters</li> <li>[11]: for Profibus only. Second set of configuration parameters</li> <li>[12]: for Profibus only. First set of user parameters The least significant byte indicates the number of parameters (possible value from 0 to 7). The remaining bytes hold the parameters</li> <li>[13]: for Profibus only. Second set of user parameters</li> <li>[14..16]: reserved</li> </ul>

## 12.146 \$FB\_SLOT: field bus slot

<i>Memory category</i>	field of fbp_tbl
<i>Load category:</i>	input/output. <i>Minor category - fieldbus</i>
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	This field contains the slot number in which the corresponding FBP board is plugged in

## 12.147 \$FB\_SL\_INIT: Field bus slave init

<i>Memory category</i>	field of fbp_tbl
<i>Load category:</i>	input/output. <i>Minor category - fieldbus</i>
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>Each slave interface has its own specific description of this field.      It must be properly initialized before using the board in the controller.      The array elements have different meanings depending on the protocol that is mounted:      For PROFIBUS:</p> <ul style="list-style-type: none"> <li>[1]: this is a bit mask where :           <ul style="list-style-type: none"> <li>– Bit 1: if set to 1, it indicates that the protocol is enabled. Otherwise the Controller assumes that the board is not present. For making operative the changes on this bit, it is needed to restart the Controller.</li> <li>– Bit 2: if set to 1, it allows to logically unlink the remote PLC from the Controller although the communication is still active. System outputs only are updated, no updating for               <ul style="list-style-type: none"> <li>– inputs. This bit is dynamically active.</li> <li>– Bit 3: reserved</li> <li>– Bit 4: if 0 the copy is done on \$DIN and \$DOUT; if 1 the copy is done of \$WORDs.</li> <li>– Bit 5: reserved</li> <li>– Bit 6: If set to 1, it indicates that the PLC has been configured for using the slave in words (2 bytes); if 0 the slave is used in single bytes (8 bits).</li> <li>– Bit 7: If 0 the word consistency is disabled; if 1 it is enabled.</li> <li>– Bit 8: If 0 the blocks consistency is disabled; if 1 it is enabled</li> <li>– Bit 9: reserved</li> <li>– Bit 10: if set to 1, the controller sends the input and output data to the remote PLC without swapping the bytes. If set to 0, the swap operation is done.</li> <li>– Bit 11..16: reserved</li> <li>– Bit 17,18,19: if the value is not 0, the modularity feature is enabled. The value of these 3 bits indicates the amount of input and output blocks to be exchanged.</li> <li>– Bit 20..32: reserved</li> </ul> </li> </ul> </li> <li>[2]: reserved</li> <li>[3]: slave address. Valid addresses 3 - 125.</li> <li>[4]: slave identification number. It is the number that identifies the controller on the PROFIBUS bus.</li> <li>[5]: if the modularity feature is enabled, this element indicated the composition of the blocks</li> <li>[6]: number of input and output bytes exchanged</li> <li>[7]: reserved</li> <li>[8]: \$WORD start index (if used)</li> <li>[9..10]: reserved</li> </ul>

For DeviceNet:

[1]: this is a bit mask where :

- Bit 1: if set to 1, it indicates that the protocol is enabled. Otherwise the Controller assumes that the board is not present. For making operative the changes on this bit, it is needed to restart the Controller.
- Bit 2: if set to 1, it allows to logically unlink the remote PLC from the Controller, although the communication is still active. System outputs only are updated, no updating on inputs. This bit is dynamically active.
- Bit 3: reserved
- Bit 4: if 0 the copy is done on \$DIN and \$DOUT; if 1 the copy is done on \$WORDS.
- Bit 5..9: reserved
- Bit 10: if set to 1, the Controller sends the input and output data to the remote PLC without swapping the bytes. If set to 0, the swap operation is done.
- Bit 11..32: reserved

[2]: baud rate. 1: 125 Kbit, 2: 250 Kbit; 3: 500 Kbit

[3]: net address (MAC ID)

[4]: vendor ID

[5]: Product Code

[6]: number of input and output bytes exchanged

[7]: reserved

[8]: \$WORD start index (if used)

[9]: Serial Number

[10]:reserved

For INTERBUS:

[1]: this is a bit mask where :

- Bit 1: if set to 1, it indicates that the protocol is enabled. Otherwise the Controller assumes that the board is not present. For making operative the changes on this bit, it is needed to restart the Controller.
- Bit 2: if set to 1, it allows to logically unlink the remote PLC from the Controller, although the communication is still active. System outputs only are updated, no updating on inputs. This bit is dynamically active.
- Bit 3: reserved
- Bit 4: if 0 the copy is done on \$DIN and \$DOUT; if 1 the copy is done of \$WORDS.
- Bit 5..9: reserved
- Bit 10: if set to 1, the Controller sends the input and output data to the remote PLC without swapping the bytes. If set to 0, the swap operation is done.
- Bit 11..32: reserved

[2]: reserved

[3]: net position

[4]: device code

[5]: reserved

[6]: number of input and output bytes exchanged

[7]: reserved

[8]: \$WORD start index (if used)

[9..10]:reserved

## 12.148 \$FB\_TYPE: Field bus type

*Memory category* field of fbp\_tbl  
*Load category:* controller. *Minor category - fieldbus*  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..4  
*S/W Version:* 1.00  
*Description:* This field contains the type of the fieldbus interface.  
It must match with the configuration found on the Controller.  
Available values are:  
– 2: Profibus  
– 3: Interbus  
– 4: DeviceNet

## 12.149 \$FDIN: Functional digital input

*Memory category* port  
*Load category:* not saved  
*Data type:* array of boolean of one dimension  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represens the functional digital input points

## 12.150 \$FDOUT: Functional digital output

*Memory category* port  
*Load category:* not saved  
*Data type:* array of boolean of one dimension  
*Attributes:* pulse usable  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the functional digital output points

## 12.151 \$FL\_ADLM: Array of delimiters

*Memory category* static

*Load category:* not saved  
*Data type:* string  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.152 \$FL\_BINARY: Text or character mode

*Memory category* static  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* none  
*S/W Version:* 1.00  
*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`

## 12.153 \$FL\_CNGF: Configuration file name

*Memory category* static  
*Load category:* controller     *Minor category - configuration*  
*Data type:* string  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It contains the name of the configuration file.

## 12.154 \$FL\_COMP: Compensation file name

*Memory category* field of arm\_data  
*Load category:* arm     *Minor category - configuration*  
*Data type:* string  
*Attributes:* none

*Limits* none  
*S/W Version:* 1.00  
*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:

## 12.155 \$FL\_DLMT: Delimiter specification

*Memory category* static  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* -1..255  
*S/W Version:* 1.00  
*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](#)

## 12.156 \$FL\_ECHO: Echo characters

*Memory category* static  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* none  
*S/W Version:* 1.00  
*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](#)

## 12.157 \$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  
*S/W Version:* 1.00  
*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](#)

## 12.158 \$FL\_PASSALL: Pass all characters

*Memory category* static  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* none  
*S/W Version:* 1.00  
*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.

## 12.159 \$FL\_RANDOM: Random file access

*Memory category* static  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* limited access, WITH OPEN FILE  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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

## 12.160 \$FL\_RDFLUSH: Flush on reading

*Memory category* static  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* limited access, WITH OPEN FILE

*Limits* none  
*S/W Version:* 1.00  
*Description:* 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.

## 12.161 \$FL\_STS: Status of last file operation

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..0  
*S/W Version:* 1.00  
*Description:* It represents the status of the last file operation undertaken by the program.

## 12.162 \$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  
*S/W Version:* 1.00  
*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](#).

## 12.163 \$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  
*S/W Version:* 1.00  
*Description:* \$FLOW\_TBL is an array of flow modulation schedules with each schedule containing the .FW\_xxx fields.

## 12.164 \$FLY\_DBUG: Cartesian Fly Debug

*Memory category*: field of crnt\_data  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..0  
*S/W Version*: 1.00  
*Description*: It contains a numerical code useful for testing the behavior of the cartesian fly. It is only used when \$FLY\_TYPE is set to FLY\_CART. The user can only reset its value to zero. All other possible values are set by the controller

## 12.165 \$FLY\_DIST: Distance in fly motion

*Memory category*: field of arm\_data  
*Load category*: arm. *Minor category* - configuration  
*Data type*: real  
*Attributes*: WITH MOVE; MOVE ALONG  
*Limits*: none  
*S/W Version*: 1.00  
*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.

## 12.166 \$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  
*S/W Version*: 1.00  
*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 moves, while on cartesian fly it has effects only if \$FLY\_TYPE is set to FLY\_NORM.

## 12.167 \$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  
*S/W Version*: 1.00  
*Description*: It indicates the meaning of \$FLY\_DIST for the trajectory planning in cartesian fly. It is only used when \$FLY\_TYPE 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

## 12.168 \$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  
*S/W Version*: 1.00  
*Description*: It represents the type of Cartesian fly motion. Valid values are represented by the predefined constants FLY\_NORMAL and FLY\_CART. FLY\_NORMAL is the default.

## 12.169 \$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  
*S/W Version*: 2.20  
*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 **Control Unit Use Manual - Chap. IO\_INST Program-I/O Configuration**.

## 12.170 \$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  
*S/W Version*: 2.20  
*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 **Control Unit Use Manual - Chap. IO\_INST Program-I/O Configuration**.

## 12.171 \$FOLL\_ERR: Following error

*Memory category*: field of crnt\_data  
*Load category*: not saved  
*Data type*: array of real of one dimension  
*Attributes*: read-only  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the current following error between the actual robot position and the desired robot position. The value is expressed in motor turns.

## 12.172 \$FUI\_DIRS: Installation path

*Memory category*: static  
*Load category*: controller     *Minor category* - environment  
*Data type*: string  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the path of directories to search when using the Filer Utility Install command

## 12.173 \$FW\_ARM: Arm under flow modulation algorithm

*Memory category*: field of \$flow\_tbl  
*Load category*: controller    *Minor category* - flow  
*Data type*: integer  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It is the arm which the FLOW MODULATE algorithm should apply to.

## 12.174 \$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  
*S/W Version*: 1.00  
*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.

## 12.175 \$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  
*S/W Version*: 1.00  
*Description*: Conversion factor. It determines the slope of the flow line. This value can be changed meanwhile the algorithm is working.

## 12.176 \$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

*S/W Version:* 1.00

*Description:* It indicates whether the Flow Modulation algorithm is currently enabled

## 12.177 \$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

*S/W Version:* 1.00

*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.

## 12.178 \$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

*S/W Version:* 1.00

*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.

## 12.179 \$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  
*S/W Version*: 1.00  
*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](#)).

## 12.180 \$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  
*S/W Version*: 1.00  
*Description*: integer flag that identifies the variable considered by the flow modulation algorithm:  
 1 for \$ARM\_VEL (cosmetic sealing),  
 2 for \$RAD\_VEL (glueing).

## 12.181 \$GEN\_OVR: General override

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: read-only  
*Limits*: 1..100  
*S/W Version*: 1.00  
*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.

## 12.182 \$GIN: Group input

*Memory category*: port  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the group input points. For further information see also [par. 5.2.3 \\$GIN and \\$GOUT on page 5-4](#) and **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.183 \$GOUT: Group output

*Memory category*: port  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the group output points. For further information see also [par. 5.2.3 \\$GIN and \\$GOUT on page 5-4](#) and **Control Unit Use** Manual - Chap. IO\_INST Program-I/O Configuration.

## 12.184 \$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  
*S/W Version*: 1.00  
*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.

## 12.185 \$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  
*S/W Version*: 1.00  
*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.

## 12.186 \$HDIN: High speed digital input

*Memory category*: port  
*Load category*: not saved  
*Data type*: array of boolean of one dimension  
*Attributes*: read-only  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the high speed digital input points present on each motion control board (DSA). 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](#). The number of elements depends on the number of DSA boards present in the system. Element 1 is related to the first DSA. Element 2 is related to the second DSA (if present)

## 12.187 \$HDIN\_SUSP: HDIN Suspend

*Memory category*: field of mcp\_data  
*Load category*: DSA  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..1  
*S/W Version*: 1.00  
*Description*: This variable is used for temporarily disabling, when set to 1, what defined for the \$HDIN by a previous call to the HDIN\_SET function. For re-enabling the \$HDIN monitoring (if it was previously enabled), \$HDIN\_SUSP should be set to 0. Note that, when [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 the call to [HDIN\\_SET Built-In Procedure](#).

## 12.188 \$HLD\_DEC\_PER: Hold deceleration percentage

*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*: 100..400  
*S/W Version*: 1.00  
*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.

## 12.189 \$HOME: Arm home position

*Memory category*: field of arm\_data  
*Load category*: arm. *Minor category* - configuration  
*Data type*: jointpos  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the home position of an arm.

## 12.190 \$IN: IN digital

*Memory category*: port  
*Load category*: not saved  
*Data type*: array of boolean of one dimension  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 1.00  
*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 5-4](#) and [Control Unit Use Manual - Chap. IO\\_INST Program-I/O Configuration](#).

## 12.191 \$IPERIOD: Interpolator period

*Memory category*: field of mcp\_data  
*Load category*: controller  
*Data type*: integer  
*Attributes*: none  
*Limits*: 1..40  
*S/W Version*: 1.00  
*Description*: It represents the time period for the generation of the position.

## 12.192 \$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  
*S/W Version*: 1.00  
*Description*: Integer registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers.

## 12.193 \$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  
*S/W Version*: 1.00  
*Description*: Integer registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers.

## 12.194 \$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  
*S/W Version:* 1.00  
*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

## 12.195 \$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  
*S/W Version:* 1.00  
*Description:* This array represents the joint limits of the work area

## 12.196 \$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  
*S/W Version:* 1.00  
*Description:* Each bit in the INTEGER represents whether the corresponding axis is present for the arm.

## 12.197 \$JNT\_MTURN: Check joint Multi-turn

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - configuration  
*Data type:* boolean

*Attributes:* field node, WITH MOVE; MOVE ALONG  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.198 \$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  
*Limits* 1..100  
*S/W Version:* 1.00  
*Description:* 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%

## 12.199 \$JOG\_INCR\_DIST: Increment jog distance

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - configuration  
*Data type:* real  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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.

## 12.200 \$JOG\_INCR\_ENBL: Jog incremental motion

*Memory category* field of crnt\_data  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* none

<i>Limits</i>	none
<i>S/W Version:</i>	1.0
<i>Description:</i>	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.

## 12.201 \$JOG\_INCR\_ROT: Rotational jog increment

<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>S/W Version:</i>	1.00
<i>Description:</i>	It represents the evolution in orientation during cartesian moves or the rotations of axes during joint moves. The default value is 0.1 degrees

## 12.202 \$JOG\_SPD\_OVR: Jog speed override

<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>	1..100
<i>S/W Version:</i>	1.00
<i>Description:</i>	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.

## 12.203 \$JPAD\_DIST: Distance between user and Jpad

<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

*S/W Version:* 1.00  
*Description:* It is the distance between the user and the robot base. It is used for defining the Jog Pad system of reference.

## 12.204 \$JPAD\_ORNT:TP4i/WiTP Angle setting

*Memory category* field of arm\_data.  
*Load category:* arm. *Minor category - configuration*  
*Data type:* real  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It is the angle in respect with the current system of reference for the Jog Pad

## 12.205 \$JPAD\_TYPE: TP4i/WiTP Jpad modality rotational or translational

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category - configuration*  
*Data type:* integer  
*Attributes:* none  
*Limits* JPAD\_LIN..JPAD\_ROT  
*S/W Version:* 1.00  
*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).

## 12.206 \$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  
*S/W Version:* 1.00  
*Description:* Jointpos registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.207 \$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  
*S/W Version*: 1.00  
*Description*: Jointpos registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.208 \$LATCH\_CNFG: Latched alarm configuration setting

*Memory category*: static  
*Load category*: controller.     *Minor category* - environment  
*Data type*: integer  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*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: modal selector switch in T2 position.

 In order to make the error unlatchable, the corresponding bit must be set to 0.

## 12.209 \$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  
*S/W Version*: 1.00  
*Description*: It represents the maximum linear acceleration expressed in meters per second squared.

## 12.210 \$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  
*S/W Version*: 1.00  
*Description*: It represents the maximum linear deceleration expressed in meters per second squared.

## 12.211 \$LIN\_SPD: Linear 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  
*S/W Version*: 1.00  
*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

## 12.212 \$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  
*S/W Version*: 1.00  
*Description*: It represents the maximum linear speed expressed in meters per second.

## 12.213 \$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:* none  
*Limits* the maximum value is  $(\$LIN\_SPD\_LIM / \$LIN\_SPD) * 100$   
*S/W Version:* 3.1  
*Description:* This predefined variable is used for changing the linear speed override during the execution of linear and circular motions. It immediately takes effect, 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**.

## 12.214 \$LOG\_TO\_DSA: Logical to physical DSA relationship

*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  
*S/W Version:* 2.4x  
*Description:* it represents the mapping between the axes of the arm and the physical channels on the DSA board.

## 12.215 \$LOG\_TO\_PHY: Logical to physical relationship

*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  
*S/W Version:* 1.00  
*Description:* It represents the mapping between the arm axes and the physical channels on the servo control board.

## 12.216 \$MAIN\_JNTP: PATH node main jointpos destination

*Memory category* static

*Load category:* not saved  
*Data type:* jointpos  
*Attributes:* limited access, field node  
*Limits* none  
*S/W Version:* 1.00  
*Description:* See the description of [\\$MAIN\\_POS: PATH node main position destination](#)

## 12.217 \$MAIN\_POS: PATH node main position destination

*Memory category* static  
*Load category:* not saved  
*Data type:* position  
*Attributes:* limited access, field node  
*Limits* none  
*S/W Version:* 1.00  
*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).

## 12.218 \$MAIN\_XTND: PATH node main xtndpos destination

*Memory category* static  
*Load category:* not saved  
*Data type:* xtndpos  
*Attributes:* limited access, field node  
*Limits* none  
*S/W Version:* 1.00  
*Description:* See the description of [\\$MAIN\\_POS: PATH node main position destination](#)

## 12.219 \$MAN\_SCALE: Manual scale factor

*Memory category* field of arm\_data  
*Load category:* arm.    *Minor category* - configuration  
*Data type:* real  
*Attributes:* privileged read-write  
*Limits* none

*S/W Version:* 1.00  
*Description:* It represents the ratio between manual and automatic speed

## 12.220 \$MCP\_BOARD: Motion Control Process board

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - configuration  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the Motion Control Process (the Motion Handler) index to which the arm belongs to.

## 12.221 \$MCP\_DATA: Motion Control Process data

*Memory category* dynamic  
*Load category:* not saved  
*Data type:* array of mcp\_data of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* \$MCP\_DATA represents the Motion Control Process data.

## 12.222 \$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  
*S/W Version:* 2.20  
*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

## 12.223 \$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  
*S/W Version:* 2.20  
*Description:* It is an array having the following meaning:  
[1]:modem initialization string

## 12.224 \$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  
*S/W Version:* 1.00  
*Description:* This is a set of data for the algorithm which controls variable acceleration and deceleration based upon the arm position.

## 12.225 \$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  
*S/W Version:* 1.00  
*Description:* Each bit in the INTEGER represents whether the corresponding axis has the modulation algorithm active

## 12.226 \$MOVE\_STATE: Move state

*Memory category* field of crnt\_data

*Load category:* not saved  
*Data type:* integer  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the current state of the motion environment. Possible values are:

- 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
- 16 : there are active moves
- 32 : there is a recovery movement in progress
- 64 : there are active moves issued in Jog mode

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

## 12.227 \$MOVE\_TYPE: Type of motion

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* field node, WITH MOVE; MOVE ALONG  
*Limits* JOINT..SEG\_VIA  
*S/W Version:* 1.00  
*Description:* 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. The default value is JOINT

## 12.228 \$MTR\_ACC\_TIME: Motor acceleration 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  
*S/W Version:* 1.00  
*Description:* It represents the minimum motor acceleration time expressed in milliseconds

## 12.229 \$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  
*S/W Version*: 1.00  
*Description*: It represents the actual motor current expressed in amperes

## 12.230 \$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  
*S/W Version*: 1.00  
*Description*: It represents the minimum motor deceleration time expressed in milliseconds.

## 12.231 \$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  
*S/W Version*: 1.00  
*Description*: It represents the maximum motor speed expressed in rotations per minute.

## 12.232 \$M\_ALONG\_1D: Internal motion control data

*Memory category*: field of mcp\_data  
*Load category*: controller.  
*Data type*: array of integer of one dimension  
*Attributes*: privileged read-write  
*Limits*: none

S/W Version:

1.00

Description:

There are certain fields of \$MCP\_DATA which do not have corresponding predefined variables, but instead can be referenced using this general array. The format and meaning of the data within these fields is reserved.

## 12.233 \$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

S/W Version:

1.00

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. C4G\_USER)

[3]: Password (e.g. C4G\_PASS)

[4]: Configuration file (e.g. C4G\_CNFG)

## 12.234 \$NET\_B\_DIR: Ethernet Boot Setup Directory

Memory category

static

Load category:

controller. *Minor category* - environment

Data type:

string

Attributes:

none

Limits

none

S/W Version:

1.00

Description:

It is the name of directory in which to find the boot file (e.g. /export/home/c4g)

## 12.235 \$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

S/W Version:

1.00

*Description:* It is an array of two elements: the first one is mapped on NET1:, the second one on NET2:  
. The bits of each element assume 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

## 12.236 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of two elements: the first one is mapped on NET1:, the second one on NET2:  
. 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'

## 12.237 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of two elements: the first one is mapped on NET1:, the second one on NET2:  
. Each element contain the name of the remote host (e.g. ibm\_server).

## 12.238 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of two elements: the first one is mapped on NET1:, the second one on NET2:. Each element contains the name of the password (e.g. Smidth).

## 12.239 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of two elements: the first one is mapped on NET1:, the second one on NET2:. Each element contains the name of the login (e.g. Pippo)

## 12.240 \$NET\_HOSTNAME: Ethernet network hostnames

*Memory category* field of board\_data  
*Load category:* retentive. *Minor category* - unique  
*Data type:* array of string of two dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* A field of BOARD\_DATA that 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)

## 12.241 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of 2 elements with the following meaning:  
[1]: Number of active servers  
[2]: reserved

## 12.242 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of 2 elements having the following meaning:  
[1]: List of hosts connected  
[2]: List of user connected

## 12.243 \$NET\_L: Ethernet Local Setup

*Memory category* field of board\_data  
*Load category:* retentive. *Minor category* - unique  
*Data type:* array of string of one dimension  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* A field of BOARD\_DATA that 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)  
[3]: Subnet mask (e.g. 255.0.0)  
[4]: IP address on backplane  
[5]: Subnet mask on backplane  
[6]: Gateway inet

## 12.244 \$NET\_MOUNT: Ethernet network mount

*Memory category*: field of board\_data  
*Load category*: retentive. *Minor category* - unique  
*Data type*: array of string of two dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: A field of BOARD\_DATA that 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:

## 12.245 \$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  
*S/W Version*: 2.20  
*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:/PPP1:  
2 = COM2:/PPP2:  
3 = COM3:/PPP3:  
4 = MDM:/PPPM:  
5 = USB Cable  
6 = USB Ethernet adaptor

## 12.246 \$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  
*S/W Version*: 2.20  
*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:/PPP1:  
 2 = COM2:/PPP2:  
 3 = COM3:/PPP3:  
 4 = MDM:/PPPM:  
 5 = USB Cable  
 6 = USB Ethernet adaptor

## 12.247 \$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  
*S/W Version*: 1.00  
*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

## 12.248 \$NET\_T\_HOST: Ethernet Network Time Protocol Host

*Memory category* static  
*Load category:* controller. *Minor category - environment*  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.3  
*Description:* Host for SNTP protocol or empty if accept incoming time setting

## 12.249 \$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  
*S/W Version:* 1.3  
*Description:* It is an array of 2 elements having the following meaning:  
[1]: Timeout of the simple network time protocol  
[2]: reserved

## 12.250 \$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  
*S/W Version:* 1.00  
*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")

## 12.251 \$NUM ALOG FILES: Number of action log files

*Memory category*: static  
*Load category*: controller.    *Minor category* - shared  
*Data type*: integer  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the number of action log files

## 12.252 \$NUM ARMS: Number of arms

*Memory category*: static  
*Load category*: arm.    *Minor category* - configuration  
*Data type*: integer  
*Attributes*: privileged read-write  
*Limits*: 1..4  
*S/W Version*: 1.00  
*Description*: It represents the number of arms present in the controller

## 12.253 \$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  
*S/W Version*: 1.00  
*Description*: It represents the number of auxiliary axes for a particular arm

## 12.254 \$NUM DEVICES: Number of devices

*Memory category*: static  
*Load category*: controller.    *Minor category* - shared  
*Data type*: integer

*Attributes:* none  
*Limits* 4..30  
*S/W Version:* 1.00  
*Description:* It represents the number of devices which can be defined at any one time. The default value is 20

## 12.255 \$NUM\_DSAS: Number of DSAs

*Memory category* static  
*Load category:* controller. *Minor category - environment*  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* 1..2  
*S/W Version:* 1.00  
*Description:* It represents the number of digital servo amplifiers (DSA) installed in the controller

## 12.256 \$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  
*S/W Version:* 1.00  
*Description:* It represents the number of joint axes for a particular arm

## 12.257 \$NUM\_LUNS: Number of LUNs

*Memory category* static  
*Load category:* controller. *Minor category - shared*  
*Data type:* integer  
*Attributes:* none  
*Limits* 1..30  
*S/W Version:* 1.00  
*Description:* It represents the maximum number of LUNs (logical unit numbers) which can be used at any one time (see [OPEN FILE Statement](#)) . The default value is 20

## 12.258 \$NUM\_MB: Number of motion buffers

*Memory category* static  
*Load category:* controller.    *Minor category - shared*  
*Data type:* integer  
*Attributes:* none  
*Limits* 5..50  
*S/W Version:* 1.00  
*Description:* It represents the number of concurrent motions which can be in the execution pipeline at the same time. The default is 10

## 12.259 \$NUM\_MB\_AHEAD: Number of motion buffers ahead

*Memory category* static  
*Load category:* controller.    *Minor category - shared*  
*Data type:* integer  
*Attributes:* none  
*Limits* 2..30  
*S/W Version:* 1.00  
*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. The default value is 4

## 12.260 \$NUM\_MCPS: Number of Motion Control Process

*Memory category* static  
*Load category:* controller  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* 1  
*S/W Version:* 1.00  
*Description:* It represents the number of Motion Control Process (Motion Handler) installed in the Controller

## 12.261 \$NUM\_PROGS: Number of active programs

*Memory category* static  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the number of active PDL2 programs at any one time

## 12.262 \$NUM\_SCRNS: Number of screens

*Memory category* static  
*Load category:* controller.     *Minor category - environment*  
*Data type:* integer  
*Attributes:* none  
*Limits* 4..12  
*S/W Version:* 1.00  
*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. The default value for this system variable is 9

## 12.263 \$NUM\_TIMERS: Number of timers

*Memory category* static  
*Load category:* controller.     *Minor category - environment*  
*Data type:* integer  
*Attributes:* none  
*Limits* 1..100  
*S/W Version:* 1.00  
*Description:* It represents the number of timers with a resolution of 10 milliseconds available at the PDL2 program level.

## 12.264 \$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  
*S/W Version*: 2.30  
*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 (.C4G) and the Controller is restarted. The default value for this predefined variable is 9.

## 12.265 \$NUM\_WEAVES: Number of weaves (WEAVE\_TBL)

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: read-only  
*Limits*: 1..16  
*S/W Version*: 1.00  
*Description*: It represents the number of weave table elements available at the PDL2 program level

## 12.266 \$ODO\_METER: average TCP space

*Memory category*: field of crnt\_data  
*Load category*: not saved  
*Data type*: real  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 2.4x

*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 and every 5 ticks in case of joint movements.  
It is zeroed in the following situations: restart, reset from pdl2 , when the variable overflows its numerical value

## 12.267 \$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  
*S/W Version:* 1.00  
*Description:* \$ON\_POS\_TBL is an array of 8 elements, with each schedule containing the \$OP\_xxx fields

## 12.268 \$OP\_JNT: On Pos jointpos

*Memory category* field of on\_pos\_tbl  
*Load category:* not saved  
*Data type:* jointpos  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.269 \$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  
*S/W Version:* 1.00

*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

## 12.270 \$OP\_POS: On Pos position

*Memory category* field of on\_pos\_tbl

*Load category:* not saved

*Data type:* position

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*Description:* This is the position associated to a \$ON\_POS\_TBL element

## 12.271 \$OP\_REACHED: On Posposition reached flag

*Memory category* field of on\_pos\_tbl

*Load category:* not saved

*Data type:* boolean

*Attributes:* read-only

*Limits* none

*S/W Version:* 1.00

*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

## 12.272 \$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

*Limits* none

*S/W Version:* 1.00

*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)

## 12.273 \$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  
*Limits* none  
*S/W Version:* 1.00  
*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)

## 12.274 \$OP\_TOOL: The On Pos Tool

*Memory category* field of on\_pos\_tbl  
*Load category:* not saved  
*Data type:* position  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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

## 12.275 \$OP\_TOOL\_DSBL: On Pos tool disable flag

*Memory category* field of on\_pos\_tbl  
*Load category:* not saved  
*Data type:* boolean

*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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

## 12.276 \$OP\_TOOL\_RMT: On Pos Remote tool flag

*Memory category* field of on\_pos\_tbl  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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

## 12.277 \$OP\_UFRAME: The On Pos Uframe

*Memory category* field of on\_pos\_tbl  
*Load category:* not saved  
*Data type:* position  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This is the User frame associated to a certain \$ON\_POS\_TBL element

## 12.278 \$ORNT\_TYPE: Type of orientation

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* field node, WITH MOVE; MOVE ALONG  
*Limits* EUL\_WORLD..RS\_TRAJ  
*S/W Version:* 1.00

*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). The default value is RS\_WORLD

## 12.279 \$OT\_COARSE: On Trajectory indicator

*Memory category* field of crnt\_data

*Load category:* not saved

*Data type:* boolean

*Attributes:* read-only

*Limits* none

*S/W Version:* 1.00

*Description:* This boolean variable indicates if the TCP (tool center point) is on the trajectory (TRUE) or not (FALSE)

## 12.280 \$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

*S/W Version:* 1.00

*Description:* This is the current position of robot joints (JOINTPOS) on the trajectory. Its value is updated any time the robot stops

## 12.281 \$OT\_POS: On Trajectory position

*Memory category* field of crnt\_data

*Load category:* not saved

*Data type:* position

*Attributes:* privileged read-write

*Limits* none

*S/W Version:* 1.00

*Description:* This is the current robot position (POSITION) on the trajectory. Its value is updated any time the robot stops

## 12.282 \$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  
*S/W Version*: 1.00  
*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

## 12.283 \$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  
*S/W Version*: 1.00  
*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

## 12.284 \$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  
*S/W Version*: 1.00  
*Description*: This is the Tool frame related to the \$CRNT\_DATA[arm\_num].OT\_POS position

## 12.285 \$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  
*S/W Version*: 1.00  
*Description*: If this variable is TRUE, the remote tool is enabled for the \$CRNT\_DATA[arm\_num].OT\_POS variable

## 12.286 \$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  
*S/W Version*: 1.00  
*Description*: This is the User frame related to the \$CRNT\_DATA[arm\_num].OT\_POS position

## 12.287 \$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  
*S/W Version*: 1.00  
*Description*: If TRUE it means that the \$CRNT\_DATA[arm\_num].OT\_POS variable is not initialised

## 12.288 \$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  
*S/W Version:* 1.00  
*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 5-4](#) and **Control Unit Use Manual** - Chap. IO\_INST Program-I/O Configuration.

## 12.289 \$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* none  
*S/W Version:* 1.00  
*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)

## 12.290 \$PGOV\_ACCURACY: required accuracy in cartesian motions

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - configuration
<i>Data type:</i>	real
<i>Attributes:</i>	WITH MOVE, MOVE ALONG
<i>Limits</i>	0.2 / 30.0
<i>S/W Version:</i>	2.4x
<i>Description:</i>	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

## 12.291 \$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  
*S/W Version*: 2.4x  
*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.

## 12.292 \$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  
*S/W Version*: 2.4x  
*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.

## 12.293 \$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  
*S/W Version*: 1.00  
*Description*: This array represents the cartesian limits of the work area

## 12.294 \$PPP\_INT: PPP Configuration

*Memory category*: static  
*Load category*: controller. *Minor category* - environment  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.20  
*Description*: It is an array having the following meaning:  
[1]: Number fo poll attempts (to check a connection)  
[2]: timeout between polls to check a connection

## 12.295 \$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  
*S/W Version*: 1.00  
*Description*: Position registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.296 \$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  
*S/W Version*: 1.00  
*Description*: Position registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.297 \$PROG\_ACC\_OVR: Program acceleration override

*Memory category:* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* WITH MOVE  
*Limits* 1..100  
*S/W Version:* 1.00  
*Description:* It represents the acceleration override in percent 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

## 12.298 \$PROG\_ARG: Program's activation argument

*Memory category:* program stack  
*Load category:* not saved  
*Data type:* STRING  
*Attributes:* none  
*Limits* none  
*S/W Version:* 2.5  
*Description:* It contains the line argument passed in when the program was activated.

## 12.299 \$PROG\_ARM: Arm of program

*Memory category:* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*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. The range of values is from 1 to 4

## 12.300 \$PROG\_CNFG: Program configuration

*Memory category* program stack

*Load category:* not saved

*Data type:* integer

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*Description:* Certain aspects of the execution of a program can be configured using this predefined variable. The bits are described below.

- 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:
  - if in PROGR state from an active program.
  - b: if in PROGR state executing the statement from the WINC4G 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 WINC4G 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 MEMORY DEBUG 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

## 12.301 \$PROG\_CONDS: Defined conditions of a program

*Memory category* program stack

*Load category:* not saved

*Data type:* array of integer of one dimension

*Attributes:* read-only

*Limits* none

<i>S/W Version:</i>	1.00
<i>Description:</i>	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. For each element of this variable, only bits 1-30 are significative (bits 31 and 32 should be ignored). Here follows an example of how this variable should be read for understanding which condition is defined for a certain program. 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

## 12.302 \$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>S/W Version:</i>	1.00
<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

## 12.303 \$PROG\_NAME: Executing program name

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	string
<i>Attributes:</i>	read-only
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	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 this. Note that the program name is stored in upper case

## 12.304 \$PROG\_SPD\_OVR: Program speed override

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved

*Data type:* integer  
*Attributes:* WITH MOVE  
*Limits* 1..100  
*S/W Version:* 1.00  
*Description:* It represents the speed override in percent for motions issued from this program. Changing this value does not affect acceleration or deceleration

## 12.305 \$PROG\_UADDR: Address of program user-defined memory access variables

*Memory category* program stack  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable has the same meaning as \$USER\_ADDR; the difference between the two variables is that there is one \$PROG\_UADDR per-program while there is only one \$USER\_ADDR in the whole system. It is used for setting the starting address in memory for \$PROG\_UBIT, \$PROG\_UBYTE, \$PROG\_UWORD and \$PROG ULONG belonging to the same program

## 12.306 \$PROG\_UBIT: Program user-defined bit memory

*Memory category* port  
*Load category:* not saved  
*Data type:* array of boolean of one dimension  
*Attributes:* pulse usable  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable has the same meaning as \$USER\_BIT and is used for accessing one bit of a memory location. The difference between \$USER\_BIT and \$PROG\_UBIT is the fact that there is one \$PROG\_UBIT per-program while there is only one \$USER\_BIT in the whole system

## 12.307 \$PROG\_UBYTE: Program user-defined byte memory

*Memory category:* port  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits*: none  
*S/W Version:* 1.00  
*Description:* This variable has the same meaning as \$USER\_BYT and is used for accessing one byte of a memory location. The difference between \$USER\_BYT and \$PROG\_UBYTE is the fact that there is one \$PROG\_UBYTE per-program while there is only one \$USER\_BYT in the whole system

## 12.308 \$PROG\_ULEN: Length of program memory access user-defined variables

*Memory category:* program stack  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits*: 0..65535  
*S/W Version:* 1.00  
*Description:* It represents the length of the memory which can be accessed using \$PROG\_UBIT, \$PROG\_UBYTE, \$PROG\_UWORD, \$PROG ULONG starting from the address contained in \$PROG\_UADDR of the same program

## 12.309 \$PROG ULONG: Program user-defined long word memory

*Memory category:* port  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits*: none  
*S/W Version:* 1.00

*Description:* This variable has the same meaning as \$USER\_LONG and is used for accessing one long word (4 bytes) of a memory location. The difference between \$USER\_LONG and \$PROG ULONG is the fact that there is one \$PROG ULONG per-program while there is only one \$USER\_LONG in the whole system

## 12.310 \$PROG\_UWORD: Program user-defined word memory

*Memory category* port  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable has the same meaning as \$USER\_WORD and is used for accessing one word (2 bytes) of a memory location. The difference between \$USER\_WORD and \$PROG\_UWORD is the fact that there is one \$PROG\_UWORD per-program while there is only one \$USER\_WORD in the whole system

## 12.311 \$PWR\_RCVR: Power failure recovery mode

*Memory category* static  
*Load category:* controller. *Minor category - shared*  
*Data type:* integer  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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. The default value for this variable is 1

## 12.312 \$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:* reaad-only  
*Limits* none  
*S/W Version:* 1.00

*Description:* It represents the ideal position in radians or millimeters. It does not take into consideration the coupling effect between axes

## 12.313 \$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

*S/W Version:* 1.00

*Description:* It represents the desired arm position expressed in radians or millimeters

## 12.314 \$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

*S/W Version:* 1.00

*Description:* It represents the current axis velocity expressed in radians per tick or millimeters per tick

## 12.315 \$RBT\_CNG: Robot board configuration

*Memory category* static

*Load category:* not saved

*Data type:* integer

*Attributes:* read-only

*Limits* none

*S/W Version:* 1.00

- 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
  - 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: PROFIBUS DP Slave interface present
  - Bit 23: InterBus-s Slave interface present
  - Bit 24: InterBus-S Master interface present
  - Bit 25..26: reserved
  - Bit 27: Device Net Slave board present
  - Bit 28..30: reserved
  - Bit 31: system in minimal configuration
  - Bit 32: reserved

## 12.316 \$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
- S/W Version:* 1.00
- Description:* It represents the family of the robot arm. The different families are:
- 1: SMART
  - 2: MAST
  - 3: reserved
  - 4: C4G (without inverse and direct kinematics)
  - 5: PMAST
  - 6..18: reserved

## 12.317 \$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  
*S/W Version:* 1.00  
*Description:* It represents the model of the robot arm. A different number is used to represent each different model of robot

## 12.318 \$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  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the COMAU name of the robot arm

## 12.319 \$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  
*S/W Version:* 1.00  
*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

## 12.320 \$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  
*Limits* none

*S/W Version:* 1.00  
*Description:* It is a bit mask used to configure specific features of some robot models. The meaning of the bits is internal

## 12.321 \$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  
*S/W Version:* 1.00  
*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

## 12.322 \$RCVR\_LOCK: Change arm state after recovery

*Memory category* field of crnt\_data  
*Load category:* not saved  
*Data type:* boolean  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.323 \$RCVR\_TYPE: Type of motion recovery

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - configuration  
*Data type:* integer  
*Attributes:* none

<i>Limits</i>	0..10
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>It represents the type of motion recovery after a motion has been interrupted. The possible values and their recovery types actions are listed below.</p> <ul style="list-style-type: none"> <li>– 0: Recovery to the interrupted trajectory with a joint interpolation.</li> <li>– 1: Recovery to the initial position of the interrupted move with a joint interpolation.</li> <li>– 2: Recovery to the final position of the interrupted move with a joint interpolation.</li> <li>– 3: Do not recover; an error message is returned to the user.</li> <li>– 4: Recovery on the interrupted trajectory, with the same move type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.</li> <li>– 5: Recovery on the initial position of the interrupted trajectory, with the same move type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.</li> <li>– 6: Recovery on the final position of the interrupted trajectory, with the same move type of the interrupted trajectory. The orientation type is WRIST_JNT if implemented, else RS_WORLD.</li> <li>– 7: Reserved.</li> <li>– 8: Reserved.</li> <li>– 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 value and the covered distance in manual motion after the robot stopping on the planned movement</li> </ul>

## 12.324 \$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>S/W Version:</i>	1.00
<i>Description:</i>	It represents the timeout in milliseconds for a READ. The default value is zero, meaning that the READ will not timeout

## 12.325 \$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>S/W Version:</i>	1.00

*Description:* 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

## 12.326 \$REF\_ARMS: Reference arms

*Memory category* field of mcp\_data  
*Load category:* controller. *Minor category* - environment  
*Data type:* integer  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It is a bit mask representing the declared arms

## 12.327 \$REMOTE: Functionality of the key in remote

*Memory category* static  
*Load category:* controller. *Minor category* - shared  
*Data type:* boolean  
*Attributes:* privileged read-write  
*Limits* none  
*S/W Version:* 1.00  
*Description:* 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.

## 12.328 \$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  
*S/W Version:* 1.00

*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 WinC4G, value 3 is for TP4i/WiTP and other values are used for external connections. Example: \$REM\_I\_STR[1,1] shows the computer that is connected via WINC4G to the controller. \$REM\_I\_STR[2,1] shows the user that is connected via WinC4G to the controller

## 12.329 \$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  
*S/W Version*: 1.00  
*Description*: This variable is used for setting internal parameters for the communication between the teach pendant and the controller

## 12.330 \$RESTART: Restart Program

*Memory category* static  
*Load category*: controller. *Minor category* - environment  
*Data type*: string  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x  
*Description*: it represents the program to be loaded and activated upon the issuing of the restart of the controller

## 12.331 \$RESTART\_MODE: Restart mode

*Memory category* static  
*Load category*: controller. *Minor category* - environment  
*Data type*: integer  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x

*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 the flag to -1.

## 12.332 \$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  
*S/W Version:* 1.00  
*Description:* It is an array of 8 elements with each element representing the definition of a restore save set. These save sets are used by the FilerUtilityRestore command, when the option /Saveset is specified, for understanding where the files should be copied from. If, for example, \$RESTORE\_SET[2] is set to 'pippo4|UD:\dir1\\*.pdl', and the command Filer Utility Restore/Saveset is issued with the pippo4 parameter for the save set, all the .pdl files that have been previously backed up from UD:\dir1 will be restored back in that directory. As a default element [1] contains the value "All|UD:.\*|s/x\*.lbe/x\*.lba"

## 12.333 \$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  
*S/W Version:* 1.00  
*Description:* It represents the maximum rotational acceleration expressed in radians per second squared.

## 12.334 \$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

*S/W Version:* 1.00

*Description:* It represents the maximum rotational deceleration expressed in radians per second squared.

## 12.335 \$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

*S/W Version:* 1.00

*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 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. In addition, if the auxiliary axis is linear, \$ROT\_SPD is not considered no matter if \$SPD\_OPT is set to SPD\_AUX1 or SPD\_AUX2 or SPD\_AUX3 or SPD\_AUX4

## 12.336 \$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

*S/W Version:* 1.00

*Description:* It represents the maximum rotational speed expressed in radians per second.

## 12.337 \$RPLC\_DATA: Data of PLC resources

*Memory category* static

*Load category:* not saved

*Data type:* array of integer of two dimension

*Attributes:* none

*Limits* none

*S/W Version:* 2.20

*Description:* This array contains data related to the PLC resource environment. The first array dimension identifies the resource; the index associated to a resource can either be viewed by the proper View command under the Resplc menu or can be read via PDL2 by means of the [RPLC\\_GET\\_IDX Built-In Procedure](#).  
 The second array dimension identifies the information of the selected resource:  
 [1]: cycle time in kernel real-time modality (default 300 microseconds)  
 [2]: cycle time in kernel cycle-to-cycle modality (default 100 microseconds)  
 [3]: cycle time in resource stopped modality (default 300 microseconds)  
 [4]: percentage of cycle time for activating external communications (default 70%)  
 [5..16]: reserved  
 At the moment, only the elements related to the first resource are used

## 12.338 \$RPLC\_STS: Status of PLC resources

*Memory category* static  
*Load category:* not saved  
*Data type:* array of integer of two dimension  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 2.20  
*Description:* This array contains data related to the PLC resource environment. The first array dimension identifies the resource; the index associated to a resource can either be viewed by the proper View command under the Resplc menu or can be read via PDL2 by means of the [RPLC\\_GET\\_IDX Built-In Procedure](#).  
 The second array dimension identifies the information of the selected resource:  
 [1]: number of cycles performed by the resource  
 [2]: duration of the last cycle of the resource (microseconds)  
 [3]: maximum duration of cycle execution (microseconds)  
 [4]: current resource state (1 for STOPPED; 2 for RUNNING)  
 [5]: resource execution modality (1 for real time; 2 for cycle to cycle)  
 [6]: number of cycles of resource overrun  
 [7..16]: reserved  
 At the moment, only the elements related to the first resource are set

## 12.339 \$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  
*S/W Version:* 1.00

*Description:* Real registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.340 \$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

*S/W Version*: 1.00

*Description:* Real registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.341 \$SAFE\_ENBL: Safe speed enabled

*Memory category* field of crnt\_data

*Load category*: not saved

*Data type*: boolean

*Attributes*: read-only

*Limits* none

*S/W Version*: 1.00

*Description:* 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 MEMORY DEBUG

## 12.342 \$SDIN: System digital input

*Memory category* port

*Load category*: not saved

*Data type*: array of boolean of one dimension

*Attributes*: privileged read-write

*Limits* none

*S/W Version*: 1.00

*Description:* It represents the system digital input points. For further information see also **par. 5.3.1 \$SDIN and \$SDOUT on page 5-6** and **Control Unit Use Manual - Chap. IO\_INST** Program-I/O Configuration.

## 12.343 \$SDOUT: 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
<i>S/W Version:</i>	1.00
<i>Description:</i>	It represents the system digital output points. For further information see also <a href="#">par. 5.3.1 \$SDIN and \$SDOUT on page 5-6</a> and <b>Control Unit Use Manual - Chap. IO_INST Program-I/O Configuration.</b>

## 12.344 \$SEG\_DATA: PATH segment data

<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>S/W Version:</i>	1.00
<i>Description:</i>	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 <a href="#">MOVE ALONG Statement</a> instead of the program data

## 12.345 \$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>S/W Version:</i>	1.00
<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

## 12.346 \$SEG\_FLY\_DIST: Parameter in segment fly motion

*Memory category*: static  
*Load category*: not saved  
*Data type*: real  
*Attributes*: limited access, field node, WITH MOVE ALONG  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: 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

## 12.347 \$SEG\_FLY\_PER: PATH segment fly percentage

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: limited access, field node, WITH MOVE ALONG  
*Limits*: 1..100  
*S/W Version*: 1.00  
*Description*: 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. The default value is 100%

## 12.348 \$SEG\_FLY\_TRAJ: Type of fly control

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: limited access, field node, WITH MOVE ALONG  
*Limits*: FLY\_AUTO..FLY\_TOL  
*S/W Version*: 1.00  
*Description*: 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

## 12.349 \$SEG\_FLY\_TYPE: PATH segment fly type

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: limited access, field node, WITH MOVE ALONG  
*Limits*: FLY\_NORM..FLY\_CART  
*S/W Version*: 1.00  
*Description*: 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

## 12.350 \$SEG\_OVR: PATH segment override

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: limited access, field node, WITH MOVE ALONG  
*Limits*: 1..100  
*S/W Version*: 1.00  
*Description*: This represents the acceleration, deceleration, and speed override for a particular PATH segment in a similar way as the \$PROG\_ACC\_OVR, \$PROG\_DEC\_OVR, and \$PROG\_SPD\_OVR 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

## 12.351 \$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  
*S/W Version*: 1.00

*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

```

## 12.352 \$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  
*S/W Version:* 1.00  
*Description:* This variable has the same purpose as \$STRESS\_PER but is used with a PATH segment. \$SEG\_STRESS\_PER does not apply to the last node of a path, \$STRESS\_PER is used instead

## 12.353 \$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  
*S/W Version:* 1.00  
*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

## 12.354 \$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  
*S/W Version:* 1.00  
*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

## 12.355 \$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  
*S/W Version:* 1.00  
*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

## 12.356 \$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  
*S/W Version*: 1.00  
*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 ly 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
  
```

## 12.357 \$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  
*S/W Version*: 1.00  
*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

## 12.358 \$SENSOR\_ENBL: Sensor Enable

*Memory category*: field of arm\_data  
*Load category*: arm. *Minor category* - sensor  
*Data type*: boolean  
*Attributes*: WITH MOVE  
*Limits*: none  
*S/W Version*: 1.00  
*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 has effect only during programmed movements unless the SENSOR\_TRK(ON) built-in has been executed

## 12.359 \$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  
*S/W Version*: 1.00  
*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

## 12.360 \$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  
*S/W Version*: 1.00  
*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

## 12.361 \$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  
*S/W Version*: 1.00  
*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

## 12.362 \$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  
*S/W Version*: 1.00

*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

## 12.363 \$SERIAL\_NUM: Serial Number

*Memory category* field of board\_data

*Load category:* not saved

*Data type:* string

*Attributes:* read-only

*Limits* none

*S/W Version:* 1.00

*Description:* This is the serial number of the board

## 12.364 \$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

*S/W Version:* 1.00

*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 respect to the tool frame or the user frame depending on \$SENSOR\_TYPE

## 12.365 \$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*: none  
*S/W Version*: 1.00  
*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

## 12.366 \$SM4C\_STRESS\_PER: Maximum Stress allowed in Cartesian SmartMove4

*Memory category*: field of arm\_data  
*Load category*: arm.    *Minor category* - configuration  
*Data type*: INTEGER  
*Attributes*: none  
*Limits*: 0..100  
*S/W Version*: 2.5  
*Description*: Cartesian SmartMove4 increases, when possible, the values of some or all move parameters (speed, acceleration and jerk) of this percentage at the maximum.

## 12.367 \$SM4\_SAT\_SCALE: SmartMove4 saturation thresholds

*Memory category*: field of arm\_data  
*Load category*: arm.    *Minor category* - configuration  
*Data type*: array of INTEGER of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x  
*Description*: It represents the limit of saturation of the current foreseen by the dynamic model in correspondence of the most significant fields; this limit is used for determining the acceleration and deceleration times with the SmartMove4 algorithm.

## 12.368 \$SPD\_OPT: Type of speed control

*Memory category*: program stack  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: field node, WITH MOVE; MOVE ALONG  
*Limits*: SPD\_JNT..SPD\_SM4C  
*S/W Version*: 1.00  
*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. The default value is SPD\_CONST

## 12.369 \$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  
*S/W Version*: 1.00  
*Description*: String registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.370 \$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  
*S/W Version*: 1.00  
*Description*: String registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

## 12.371 \$STARTUP: Startup program

*Memory category* static  
*Load category:* controller.    *Minor category - environment*  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.372 \$STARTUP\_USER: Startup user

*Memory category* static  
*Load category:* controller.    *Minor category - shared*  
*Data type:* string  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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

## 12.373 \$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  
*S/W Version:* 1.00  
*Description:* It represents the maximum stress of the arm during a cartesian fly. It is only used if \$FLY\_TYPE 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

## 12.374 \$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  
*S/W Version*: 1.00  
*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

## 12.375 \$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  
*S/W Version*: 1.00  
*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

## 12.376 \$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  
*S/W Version*: 1.00  
*Description*: It represents the system negative stroke ends of the arm. This value cannot be changed by the user

## 12.377 \$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  
*S/W Version:* 1.00  
*Description:* It represents the system positive stroke ends of the arm. This value cannot be changed by the user

## 12.378 \$SWIM\_ADDR: SWIM address

*Memory category* static  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable contains the physical address of the SWIM boards dual port memory

## 12.379 \$SWIM\_CNFG: SWIM configuration mode

*Memory category* static  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* read-only  
*Limits* none  
*S/W Version:* 1.00  
*Description:* This variable represents the configuration of the SWIM boards. Currently the following bits are defined:
 

- Bit 1: SWIM1 Board configured
- Bit 2: SWIM2 Board configured
- Bit 3: SWIM1 Board active
- Bit 4: SWIM2 Board active
- Bit 16: SWIM1 Board plugged in
- Bit 17: SWIM2 Board plugged in

## 12.380 \$SWIM\_INIT: SWIM Board initialization parameters

*Memory category* static

*Load category:* input/output. *Minor category - SWIM*  
*Data type:* array of integer of two dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* The variable contains initialization and setup information for the SWIM Board. It must be properly initialized before using the board in the controller. The first index is the SWIM board number. The meaning of the second index is:  
 [1]: this is a bit mask where :  
 Bit 1: if set to 1, it indicates that the board can be recognized by the controller.  
 – Otherwise the controller assumes that the SWIM board is not present. For making operative the changes on this bit, it is needed to issue a restart of the controller.  
 – Bit 2: SWIM board disabled via software. This means that data from the SWIM board are not copied in the \$WORD array. Viceversa, the controller output \$WORDS are copied in the SWIM board.  
 – Bit 12: If the SWIM board is disabled via software, the first \$WORD of Input data is not updated  
 [2]: index, in the \$WORD array, starting from which data are copied.  
 [3]: number of \$WORDS to be copied  
 [4]: application profile  
 [5..8]: reserved

## 12.381 \$SYNC\_ARM: Synchronized arm of program

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 1..4  
*S/W Version:* 1.00  
*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

## 12.382 \$SYS\_CALL\_OUT: Output lun for SYS\_CALL

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00

*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

## 12.383 \$SYS\_CALL\_STS: Status of last SYS\_CALL

*Memory category* program stack

*Load category*: not saved

*Data type*: integer

*Attributes*: none

*Limits* 0..0

*S/W Version*: 1.00

*Description*: It represents the last error that occurred executing a SYS\_CALL by the program

## 12.384 \$SYS\_CALL\_TOUT: Timeout for SYS\_CALL

*Memory category* program stack

*Load category*: not saved

*Data type*: integer

*Attributes*: none

*Limits* 0..MAXINT

*S/W Version*: 1.00

*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

## 12.385 \$SYS\_ERROR: Last system error

*Memory category* static

*Load category*: not saved

*Data type*: integer

*Attributes*: none

*Limits* 0..0

*S/W Version*: 1.00

*Description*: It represents the last error that occurred in the system

## 12.386 \$SYS\_ID: Robot System identifier

<i>Memory category</i> :	field of board_data
<i>Load category</i> :	retentive. <i>Minor category</i> - unique
<i>Data type</i> :	string
<i>Attributes</i> :	privileged read-write
<i>Limits</i>	none
<i>S/W Version</i> :	1.00
<i>Description</i> :	A field of BOARD_DATA that contains information about the controller's id. Every controller has a unique system identifier that should not be changed. This identifier is used for locating controller specific files eg. <\$SYS_ID>.C4G for configuration file

## 12.387 \$SYS\_INP\_MAP: Configuration for system bits in Input on fieldbuses

<i>Memory category</i> :	static
<i>Load category</i> :	input/output. <i>Minor category</i> - fieldbus
<i>Data type</i> :	array of integer of two dimension
<i>Attributes</i> :	privileged read-write
<i>Limits</i>	none
<i>S/W Version</i> :	1.00
<i>Description</i> :	This table contains information about the mapping of the \$SDIN on the system word in input on the primary fieldbus. For further information see also <b>Control Unit Use Manual</b> - Chap. IO_INST Program-I/O Configuration.

## 12.388 \$SYS\_OUT\_MAP: Configuration for system bits in Output on fieldbuses

<i>Memory category</i> :	static
<i>Load category</i> :	input/output. <i>Minor category</i> - fieldbus
<i>Data type</i> :	array of integer of two dimension
<i>Attributes</i> :	privileged read-write
<i>Limits</i>	none
<i>S/W Version</i> :	1.00
<i>Description</i> :	This table contains information about the mapping of the \$SDOUT on the system word in output on the primary fieldbus. For further information see also <b>Control Unit Use Manual</b> - Chap. IO_INST Program-I/O Configuration.

## 12.389 \$SYS\_PARAMS: Robot system identifier

*Memory category*: field of board\_data  
*Load category*: retentive. *Minor category* - unique, parameter  
*Data type*: array of integer of one dimension  
*Attributes*: privileged read-write  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: There are certain fields of \$BOARD\_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. It is recommended not to modify the data in this array!

## 12.390 \$SYS\_STATE: State of the system

*Memory category*: static  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: read-only  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents a complete description of the state of the system and consists of the following mask:

- 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 WINC4G connected on a PC
- Bit 6: Model of Teach Pendant (see also bit 23)
- Bit 7: reserved
- Bit 8: State selector on PROG-T2
- 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:
  - 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 ON
- Bit 32: Reserved

## 12.391 \$TERM\_TYPE: Type of motion termination

*Memory category:* program stack

*Load category:* not saved

*Data type:* integer

*Attributes:* WITH MOVE; MOVE ALONG

*Limits* FINE..JNT\_COARSE

*S/W Version:* 1.00

*Description:* 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.

- 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.
- 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[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[axis\_num]

## 12.392 \$THRД\_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>S/W Version:</i>	1.00
<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 \$THRД_CEXP for each thread of execution. The user is only allowed to set this variable to 0.</p> <p>For example:</p> <pre> WAIT FOR HOLD OR EVENT 94 OR (\$FDIN[5]=TRUE) AND (\$FDOUT[5]=FALSE) IF \$THRД_CEXP = 3 THEN     WRITE ('\$FDIN[5] IS TRUE AND \$FDOUT[5] IS FALSE', NL) ELSE     IF \$THRД_CEXP = 2 THEN         WRITE ('EVENT 94 event triggered', NL)     ELSE -- \$THRД_CEXP is 1         WRITE ('The HOLD event triggered', NL)     ENDIF ENDIF </pre> <p>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 <a href="#">WAIT FOR Statement</a>, \$THRД_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</p>

## 12.393 \$THRД\_ERROR: Error of each thread of execution

<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>S/W Version:</i>	1.00

*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

## 12.394 \$THRД\_PARAM: Thread Parameter

*Memory category* program stack

*Load category:* not saved

*Data type:* integer

*Attributes:* read-only

*Limits* none

*S/W Version:* 1.00

*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 \$THRД\_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 is used for many different types of events

## 12.395 \$TIMER: Clock timer

*Memory category* port

*Load category:* not saved

*Data type:* array of integer of one dimension

*Attributes:* none

*Limits* none

*S/W Version:* 1.00

*Description:* It represents 1 msec timers with resolution of 10 msecs available for use in PDL2 programs. The number of elements is dependant on the predefined variable \$NUM\_TIMERS. Timer elements can be attached by different programs. See [ATTACH Statement](#) and [DETACH Statement](#)

## 12.396 \$TOL\_ABТ: Tolerance anti-bounce time

*Memory category* field of arm\_data

*Load category:* arm.    *Minor category - configuration*

*Data type:* integer

**Attributes:** none  
**Limits** 0..2000  
**S/W Version:** 1.00  
**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. The default value is 0 milliseconds

## 12.397 \$TOL\_COARSE: Tolerance coarse

**Memory category** field of arm\_data  
**Load category:** arm. *Minor category - configuration*  
**Data type:** real  
**Attributes:** WITH MOVE; MOVE ALONG  
**Limits** none  
**S/W Version:** 1.00  
**Description:** It represents the spherical region, in millimeters, for defining the termination of motion when \$TERM\_TYPE is set to COARSE. The default value depends on the kind of robot

## 12.398 \$TOL\_FINE: Tolerance fine

**Memory category** field of arm\_data  
**Load category:** arm. *Minor category - configuration*  
**Data type:** real  
**Attributes:** WITH MOVE; MOVE ALONG  
**Limits** none  
**S/W Version:** 1.00  
**Description:** It represents the spherical region, in millimeters, for defining the termination of motion when \$TERM\_TYPE is set to FINE. The default value depends on the kind of robot

## 12.399 \$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  
**S/W Version:** 1.00

*Description:* It represents the tolerance in degrees for each rotating joint and in millimeters for each translating joint. This parameter is used when \$TERM\_TYPE 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.

## 12.400 \$TOL\_JNT\_FINE: 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  
*S/W Version:* 1.00  
*Description:* It represents the tolerance in degrees for each rotating joint and in millimeters for each translating joint. This parameter is used when \$TERM\_TYPE 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

## 12.401 \$TOL\_TOUT: Tolerance timeout

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - configuration  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..20000  
*S/W Version:* 1.00  
*Description:* 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. The default value depends on the kind of robot

## 12.402 \$TOOL: Tool of arm

*Memory category* field of arm\_data  
*Load category:* arm. *Minor category* - chain  
*Data type:* position  
*Attributes:* WITH MOVE; MOVE ALONG  
*Limits* none  
*S/W Version:* 1.00  
*Description:* It represents the tool dimension and orientation

## 12.403 \$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  
*S/W Version:* 1.00  
*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)

## 12.404 \$TOOL\_FRICTION: Tool Friction

*Memory category:* field of arm\_data  
*Load category:* arm. *Minor category - chain*  
*Data type:* array of real of one dimension  
*Attributes:* privileged read-write, WITH MOVE  
*Limits*: none  
*S/W Version:* 1.00  
*Description:* This array represents the static and viscous frictions of wrist axes of the robot determined by the algorithm of load identification

## 12.405 \$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  
*S/W Version:* 1.00  
*Description:* This array represents the tensor of inertia of the tool determined by the algorithm of load identification

## 12.406 \$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  
*S/W Version*: 1.00  
*Description*: It represents the tool mass expressed in Kg

## 12.407 \$TOOL\_RMT: Fixed Tool

*Memory category*: field of arm\_data  
*Load category*: arm. *Minor category* - chain  
*Data type*: boolean  
*Attributes*: WITH MOVE  
*Limits*: none  
*S/W Version*: 1.00  
*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

## 12.408 \$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  
*S/W Version*: 2.4x  
*Description*: it represents the more far away point reached by the tool: if not implicitly declared, it is the same by default of \$TOOL

## 12.409 \$TP\_ARM: Teach Pendant current arm

*Memory category*: static  
*Load category*: not saved

*Data type:* integer  
*Attributes:* read-only  
*Limits* 1..4  
*S/W Version:* 1.00  
*Description:* It represents the arm currently selected on the teach pendant and displayed on the status line of the system screen

## 12.410 \$TP\_GEN\_INCR: Incremental value for general override

*Memory category* static  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..10  
*S/W Version:* 1.00  
*Description:* It represents the general override increment currently being used at the teach pendant

## 12.411 \$TP\_MJOG: Type of TP jog motion

*Memory category* static  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..3  
*S/W Version:* 1.00  
*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

## 12.412 \$TP\_ORNT: Orientation for jog motion

*Memory category* field of arm\_data  
*Load category:* arm.    *Minor category* - configuration  
*Data type:* integer  
*Attributes:* none

<i>Limits</i>	WRIST_JNT..RPY_WORLD
<i>S/W Version:</i>	1.00
<i>Description:</i>	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

## 12.413 \$TP\_SYNC\_ARM: Teach Pendant's synchronized arms

<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>	0..4
<i>S/W Version:</i>	3.00
<i>Description:</i>	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).
<i>Examples:</i>	<b>Arm: 2&lt;1</b> means that Arm 2 is the main Arm and it is selected for jogging <b>Arm: 1&lt;2</b> means that Arm 1 is the main Arm and it is selected for jogging.

For further information about how to select an Arm, see **Control Unit Use Manual, par. 6.5.1.4 Right Menu.**

## 12.414 \$TUNE: Internal tuning parameters

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category - shared</i>
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	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, DSA 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 WinC4G 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 TP4i/WiTP backlight. The default value is 180 seconds (3 minutes).
  - Allowed values:
  - 1 - the TP4i/WiTP will never switch;
  - 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..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

## 12.415 \$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>	field node, WITH MOVE; MOVE ALONG

<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<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

## 12.416 \$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>S/W Version:</i>	1.00
<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

## 12.417 \$UFRAME: User frame of an arm

<i>Memory category</i>	field of arm_data
<i>Load category:</i>	arm. <i>Minor category</i> - chain
<i>Data type:</i>	position
<i>Attributes:</i>	WITH MOVE
<i>Limits</i>	none
<i>S/W Version:</i>	1.00
<i>Description:</i>	It represents the location and orientation of the user frame

## 12.418 \$USER\_ADDR: Address of user-defined variables

<i>Memory category</i>	static
<i>Load category:</i>	controller. <i>Minor category</i> - vars
<i>Data type:</i>	array of integer of one dimension
<i>Attributes:</i>	none
<i>Limits</i>	none

*S/W Version:* 1.00  
*Description:* Used to set the starting address in memory of \$USER\_BIT, \$USER\_BYTE, \$USER\_WORD and \$USER\_LONG

## 12.419 \$USER\_BIT: User-defined bit memory

*Memory category* port  
*Load category:* not saved  
*Data type:* array of boolean of one dimension  
*Attributes:* pulse usable  
*Limits* none  
*S/W Version:* 1.00  
*Description:* Used for accessing bits of memory at an address defined by the user. The starting address from which bits are counted is specified in \$USER\_ADDR[1]. Note that the bits numbering considers the byte in memory to which each bit belongs. For example, \$USER\_BIT[4] is the 4th bit, starting from the right, inside the first byte respect to the address specified in \$USER\_ADDR[1]; \$USER\_BIT[10] is the 2nd bit, starting from the right, of the second byte respect to the address specified in \$USER\_ADDR[1]

## 12.420 \$USER\_BYTE: User-defined byte memory

*Memory category* port  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*Description:* Used for accessing memory as a byte at an address defined by the user. The starting address from which bytes are counted is specified in \$USER\_ADDR[2]

## 12.421 \$USER\_LEN: Length of User-defined variables

*Memory category* static  
*Load category:* controller. *Minor category - vars*  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits* 0..65535  
*S/W Version:* 1.00

*Description:* It represents the length of the memory which can be accessed using \$USER\_BIT, \$USER\_BYTE, \$USER\_WORD, \$USER\_LONG starting from the address contained in \$USER\_ADDR

## 12.422 \$USER\_LONG: User-defined long word memory

*Memory category* port  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: Used for accessing memory as a long word (4 bytes) at an address defined by the user. The starting address from which longwords are counted is specified in \$USER\_ADDR[4]

## 12.423 \$USER\_WORD: User-defined word memory

*Memory category* port  
*Load category*: not saved  
*Data type*: array of integer of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: Used for accessing memory as a word (2 bytes) at an address defined by the user. The starting address from which words are counted is specified in \$USER\_ADDR[3]

## 12.424 \$VERSION: Software version

*Memory category* static  
*Load category*: not saved  
*Data type*: real  
*Attributes*: read-only  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: It represents the current version of the system software

## 12.425 \$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  
*S/W Version*: 3.1  
*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**).

## 12.426 \$VP2\_TOUT: Timeout value for asynchronous VP2 requests

*Memory category*: program stack  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..MAXINT  
*S/W Version*: 2.30  
*Description*: This variable contains the timeout value for asynchronous VP2 calls with a value of 0 meaning an indefinite wait.

## 12.427 \$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  
*S/W Version*: 2.4x  
*Description*: This variable is used for setting Visual PDL2 internal parameters.

## 12.428 \$WEAVE\_MODALITY: Weave modality

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integere
<i>Attributes:</i>	field node, WITH MOVE; MOVE ALONG
<i>Limits</i>	none
<i>S/W Version:</i>	1.20
<i>Description:</i>	<p>This variable identifies the modality in which the shape of the wave is generated.</p> <ul style="list-style-type: none"> <li>– Setting it to 0 (default value), the wave shape is created using the traversal speed;</li> <li>– setting it to 1, the wave shape is created using the length of the weave.</li> </ul> <p>It is not effected by changing other parameters (override, speed, etc.).</p>

## 12.429 \$WEAVE\_MODALITY\_NOMOT: Weave modality (only for no arm motion)

<i>Memory category</i>	field of crnt_data
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	none
<i>Limits</i>	none
<i>S/W Version:</i>	3.1
<i>Description:</i>	<p>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.</p>

## 12.430 \$WEAVE\_NUM: Weave table number

<i>Memory category</i>	program stack
<i>Load category:</i>	not saved
<i>Data type:</i>	integer
<i>Attributes:</i>	field node, WITH MOVE; MOVE ALONG
<i>Limits</i>	0..10
<i>S/W Version:</i>	1.00
<i>Description:</i>	<p>It represents the weave table (\$WEAVE_TBL) element to be used in the next motions. The value of zeero disables the weaving</p>

## 12.431 \$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  
*S/W Version:* 3.1  
*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.

## 12.432 \$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  
*S/W Version:* 2.4x  
*Description:* it contains the index of the active weaving phase

## 12.433 \$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  
*S/W Version:* 1.00  
*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

## 12.434 \$WEAVE\_TYPE: Weave type

*Memory category*: program stack  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: field node, WITH MOVE; MOVE ALONG  
*Limits*: 0..10  
*S/W Version*: 1.00  
*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

## 12.435 \$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  
*S/W Version*: 3.1  
*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.

## 12.436 \$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  
*S/W Version*: 1.00  
*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

## 12.437 \$WFR\_TOUT: Timeout on a WAIT FOR

*Memory category:* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..2147483647  
*S/W Version:* 1.00  
*Description:* It represents the timeout in milliseconds for a WAIT FOR. The default value is zero, meaning that the WAIT FOR will not timeout

## 12.438 \$WORD: PLC WORD data

*Memory category* port  
*Load category:* not saved  
*Data type:* array of integer of one dimension  
*Attributes:* none  
*Limits* none  
*S/W Version:* 1.00  
*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 5-16](#) and **Control Unit Use Manual - Chap. IO\_INST Program-I/O Configuration.**

## 12.439 \$WRITE\_TOUT: Timeout on a WRITE

*Memory category* program stack  
*Load category:* not saved  
*Data type:* integer  
*Attributes:* none  
*Limits* 0..2147483647  
*S/W Version:* 1.00  
*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

## 12.440 \$WV\_AMP\_PER: Weave amplitude percentage

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: WITH MOVE  
*Limits*: 0..1000  
*S/W Version*: 1.00  
*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. The default value is 100%

## 12.441 \$WV\_CNTR\_DLW: Weave center dwell

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0.10000  
*S/W Version*: 1.00  
*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.

## 12.442 \$WV\_END\_DLW Weave end dwell

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: Integer  
*Attributes*: none  
*Limits*: 0 10000  
*S/W Version*: 2.4x  
*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.

## 12.443 \$WV\_LEFT\_AMP: Weave left amplitude

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: real  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*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

## 12.444 \$WV\_LEFT\_DWL: Weave left dwell

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..10000  
*S/W Version*: 1.00  
*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.

## 12.445 \$WV\_LENGTH\_WAVE: Wave length

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..10000  
*S/W Version*: 1.200  
*Description*: It represents the length, in millimeters, of the weaving shape wave

## 12.446 \$WV\_ONE\_CYCLE: Weave one cycle

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: boolean  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x  
*Description*: If set to TRUE the weave will only perform one cycle

## 12.447 \$WV\_PLANE: Weave plane angle

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: real  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*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

## 12.448 \$WV\_RIGHT\_AMP: Weave right amplitude

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: real  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*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

## 12.449 \$WV\_RIGHT\_DWL: Weave right dwell

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: integer  
*Attributes*: none  
*Limits*: 0..10000  
*S/W Version*: 1.00  
*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.

## 12.450 \$WV\_SMOOTH: Weave smooth enabled

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: boolean  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*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

## 12.451 \$WV\_SPD\_PROFILE: Weave speed profile enabled

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: boolean  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x  
*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.

## 12.452 \$WV\_TRV\_SPD: Weave transverse speed

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: real  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: 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

## 12.453 \$WV\_TRV\_SPD\_PHASE: Weave transverse speed phase

*Memory category*: field of weave\_tbl  
*Load category*: not saved  
*Data type*: array of real of one dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 2.4x  
*Description*: if \$WEAVE\_MODALITY is set to 0, it hasn't effect. If \$WEAVE\_MODALITY is set to 1, it represents 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

## 12.454 \$XREG: Xtndpos registers - saved

*Memory category*: statics  
*Load category*: controller.    *Minor category - vars*  
*Data type*: array of xtndpos of two dimension  
*Attributes*: none  
*Limits*: none  
*S/W Version*: 1.00  
*Description*: Xtndpos registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved register

## 12.455 \$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

*S/W Version:* 1.00

*Description:* Xtndpos registers are "" variables that can be used by users instead of having to define variables. For the different datatypes of Integer, Real, String, Boolean, Jointpos, Position & Xtndpos there are saved and non-saved registers

# 13. POWER FAILURE RECOVERY

This appendix describes the features available in the PDL2 programming language to detect and recover from a power failure. These features should allow an application program to gracefully recover from a power failure and continue exactly where the power failure occurred.

The following list describes the behavior of the system during a power failure and the PDL2 language components that can be used to detect and recover from the failure.

- the power failure recovery performance is enabled by default. To disable it, \$PWR\_RCVR system variable is to be set to FALSE:

```
$PWR_RCVR := FALSE
```

- while power failure recovery is enabled, the startup program will not be executed every time the power switch is turned on. The name of the startup program is stored in the \$STARTUP predefined variable;
- a PDL2 program can detect the power failure recover event by using the WHEN POWERUP DO event in a condition handler. This event is triggered only when the system restarts in power failure recovery mode. It will not be triggered when the system restarts due to a CONFIGURE CNTRLER RESTART COLD command. Refer to the [Chap.8. - Condition Handlers](#) for more information on the POWERUP event and the use of condition handlers;
- when a power failure occurs, the HOLD system event and the DRIVES turned OFF events are triggered. A PDL2 program can use these events to perform the same actions that are normally performed during a HOLD and DRIVES turned OFF events;
- when the DRIVES are turned ON, it is possible that the system will log some errors from the RSM board. These errors are caused by the 24 volt supply disappears before the SMP board power supply, which causes the alarm to be read. The system can only recover from this condition after it is restarted. The system can not delete the alarm logged in the ERROR.LOG file and displayed on the screen;
- a READ statement will return an error if it is pending on the TP or a serial communications line;
- a WRITE statement to a serial communications line will return an error. A WRITE statement to the TP will not return an error and the window will contain the correct display data after the system is restarted;
- some of the devices in the system require several seconds to perform their diagnostic and startup procedures. While these operations are being performed, the PDL2 program will begin to execute. Some of the PDL2 language statements and built-in routines will fail if they are executed before these devices have completed their startup procedures. Built-in routines, like the ARM\_POS built-in, or pressing the DRIVE ON or START button will fail if the startup procedures have not been completed. A PDL2 program should wait until the power failure recovery has completed before continuing execution. To cause a PDL2 program to wait until the power failure recover has completed, create a condition handler that triggers on the POWERUP event and then waits until bit number 9 of the \$SYS\_STATE predefined variable is cleared. This procedure is shown in an example program at the end of this appendix;

- removing boards from the rack when the Controller is switched off can cause problems during recovery from power failure at the system restart. The system cannot support all deriving problems;



**The user is strongly recommended NOT TO DO such an operation!**

- if a certain board must be extracted from the rack for changing the system configuration, it is recommended to issue a CONFIGURE CONTROLLER RESTART COLD command and then, without waiting the restart of the controller, the power should be switched off using the electronics power switch on the Operator Panel. At this point the specific board can be removed and the system can be restarted using the electronic switch;
- in this way, the power failure recovery is not performed by the controller that accepts the new configuration, eventually signalling the absence of the previously configured board(s);



**It is also suggested to check that the TP4i/WiTP Teach Pendant is connected after the power failure recovery.**

**Please use the proper event (EVENT 116) inside a CONDITION or a WAIT FOR statement.**

The following PDL2 program demonstrate how to recover from a power failure during a READ or WRITE statement.

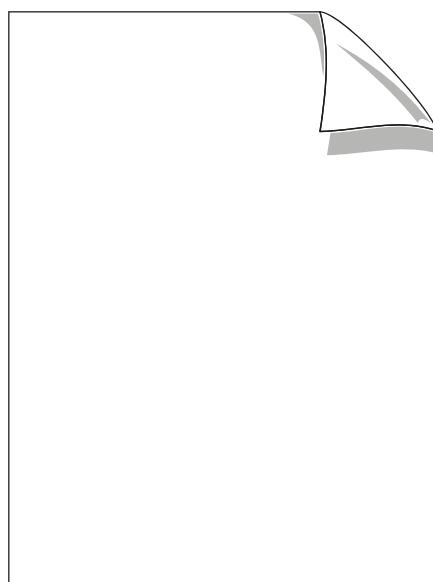
```

PROGRAM pf_read NOHOLD
VAR rs232 : INTEGER NOSAVE
  s : STRING[10] NOSAVE
BEGIN
  OPEN FILE rs232 (COM1:, RW)
    ERR_TRAP_ON(39990) -- "READ/Filer error 11294" (11294 --> "power
failure")
CYCLE
  READ rs232 (s) -- input character.
  IF $FL_STS = 0 THEN
    WRITE (s) -- Data is valid.
  ENDIF
END pf_read
PROGRAM pf_write NOHOLD
VAR rs232 : INTEGER NOSAVE
BEGIN
  OPEN FILE rs232 (COM1:, RW)
    ERR_TRAP_ON(40033) -- "WRITE error 11294" (11294 --> "power
failure")
    ERR_TRAP_ON(39990) -- "READ/Filer error 11294"
CYCLE
  WRITE rs232 ($CYCLE)
END pf_write

```

Waiting until the power failure recovery is complete is shown in the next example program.

```
PROGRAM pf_sync NOHOLD
VAR a : POSITION NOSAVE
ROUTINE pf_wait_until_complete
BEGIN
    $FDOUT[5] := ON -- turn on U1 lamp
    WAIT FOR BIT_TEST($SYS_STATE, 9, OFF) -- wait for PF to
complete
    $FDOUT[5] := OFF -- turn off U1 lamp
END pf_wait_until_complete
BEGIN
    CONDITION[1] NODISABLE :      wait for POWERUP after PF.
    WHEN POWERUP DO
        pf_wait_until_complete -- wait until power fail recover is
complete
    ENDCONDITON
    ENABLE CONDITION[1] -- Enable PF monitoring.
CYCLE
    WAIT FOR $FDIN[5] -- Just wait... (press U1 to print position)
    a := ARM_POS -- Get ARM position; if done before the end of
    WRITE (a) -- recovery system show "arm not cal." warning.
END pf_sync
```



# 14. TRANSITION FROM C3G TO C4G 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 C4G Control Unit a PDL2 program written in the past for C3G Control Unit.

Detailed information is supplied about the following topics:

- [System parameters values](#)
- [Terminology](#)
- [Old programs compatibility with the new Controller Unit](#)
- [Devices and Directories](#)
- [System Variables](#)
- [Predefined constants](#)
- [Built-in routines and functions](#)
- [Open File Statement](#)
- [Conditions](#)
- [Swapping to a foreign language](#)
- [System menu Commands](#)
- [Compressed files](#)
- [Error and action Logging](#)
- [PC interface](#)
- [Fieldbuses configuration](#)
- [MMUX \(motor multiplexer\)](#)

## 14.2 System parameters values

The maximum number of joints is : 10. On C3G Controller it was 8.

Maximum string length: 1024. On C3G Controller it was 254.

Maximum length of loadable program name: 32 characters. On C3G Controller it was 8.

Maximum length for a file name specification, including full path name: 128 characters

The default dimension of the program stack is 1000 bytes. On C3G controller it was 300 bytes.

## 14.3 Terminology

SCC board is now identified by the Motion Control Process.

RBC board is now identified with SMP board.

The main storage device is now UD: it corresponds to RD: of C3G Controller.

FD: is no more handled. If using the disk on key, it should be changed with XD:

## 14.4 Old programs compatibility with the new Controller Unit

To run on C4G Controller a PDL2 program that has been developed on C3G Controller, it is necessary to start from ASCII format, therefore from .PDL and .LSV files.

.COD and .VAR files generated on C3G Controller cannot be loaded and run on C4G Controller. Please start from ASCII format (.PDL and .LSV) before copying them on C4G Controller.

If the user only has the binary (.COD and .VAR) file, it is necessary to translate it, either on C3G controller or on a PC using the PDL2 offline translator delivered for C3G Controller, in order to obtain the corresponding .PDL file.

Then the program (.PDL and .LSV) can be checked for C4G Controller, either translating it directly on the Controller or using WINC4G program.



**It is however recommended to read the current chapter because there can be some differences that are not necessarily detected by the translation phase.**

## 14.5 Devices and Directories

The main storage devices is UD: ; on the C3G it was RD:.

The main secondary storage device is XD: ; on the C3G it was FD:.

These are now handled on the C4G controller. The default directory is UD:. Use the DIR\_SET built-in procedure for defining the default directory to be used within a PDL2 program.

There is a default directory specification for each device (programming unit, terminal of winc4g and sys\_call) from which commands are issued.

For changing the directory to be accessed by a PDL2 program, the DIR\_SET built-in routine must be called. For changing the directory at the command level, the FilerUtilityDirectory Change command must be issued from the device being used (programming unit or winc4g); SYS\_CALL cannot be applied to this command.

Please refer to the description of [\\$DFT\\_DV: Default devices](#) System Variable in [Chap. Predefined Variables List](#).

Device and directories follow the MSDOS model: <device>:<directory>\<filename> (e.g. UD:\weld1\prog1.cod). The maximum level for a directory is 8.

Some examples follow, assuming that the current directory is 'UD:\dir1':

```
OPEN FILE vi_lun ('progone') --will open 'UD:\dir1\progone'.
OPEN FILE vi_lun ('\dir2\prog2') --will open 'UD:\dir2\prog2'
OPEN FILE vi_lun ('TD:\dir3\prog3') -- will look for
-- 'TD:\dir3\prog3'
OPEN FILE vi_lun ('..\prog4') -- if the default selected directory
-- is 'TD:\dir3', the
-- 'TD:\prog4' file will be opened.
```

## 14.6 System Variables

Information are supplied about the following topics:

- [File <\\$SYS\\_ID>.C4G](#)
- [Removed System Variables](#)
- [Variables which have been renamed](#)
- [\\$FDIN, \\$FDOUT, \\$SDIN, \\$SDOUT](#)
- [BIT referenced System Variables](#)
- [Array element referenced System Variables](#)
- [Miscellaneous](#)

### 14.6.1 File <\$SYS\_ID>.C4G

System variables are stored in a file called <SYS\_ID>.C4G that are stored in the system directory UD:\sys. This file was C3G.SYS, on C3G Controller. A C3G.SYS file cannot be loaded on a C4G Controller.

### 14.6.2 Removed System Variables

The following system variables have been removed:

- \$A\_ABYTE
- \$A\_AWORD
- \$C\_ABYTE1
- \$ARM\_LOADED
- \$CAL\_FILE
- \$CNTRL\_NUM
- \$CNTRL\_OPT
- \$DEVNET\_INIT
- \$DRV\_ADDR
- \$D\_AWORD1
- \$HOUR\_METER
- \$IBSSL\_INIT
- \$IDIN
- \$IDOUT
- \$LEN\_ELOG

- \$MASTER\_INIT
- \$MTR\_RES
- \$NET\_S\_STR
- \$NIO\_INIT,
- \$OFF\_AREA
- \$OFF\_LIM\_MAX
- \$OFF\_LIM\_MIN
- \$RIO\_INIT
- \$PROFDP\_INIT
- \$NUM\_ELOG
- \$NUM\_MB( see \$NUM\_DSAS)
- \$PIO\_GINTBL, \$PIO\_GOUTTBL, \$PIO\_LOGTBL, \$PLCSTW
- \$PWR\_ADDR
- \$RPT\_ADDR
- \$RESOLVER
- \$RB\_MARK
- \$S\_ALONG1
- \$SYS\_PROT
- \$SYS\_PROT\_STATE
- \$SWIM\_INIT
- \$WORK\_AREA

### 14.6.3 Variables which have been renamed

- \$SC\_DATA has been renamed in \$MCP\_DATA.
- \$DSP\_DATA has been renamed in \$DSA\_DATA
- \$SC\_BOARD has been renamed in \$MCP\_BOARD
- \$STRK\_END\_SYS : it was a bi-dimensional system variable. On C4G Controller it no longer exists and has been replaced with 2 mono-dimensional arrays: \$STRK\_END\_SYS\_P , \$STRK\_END\_SYS\_N which represent the positive and negative stroke ends of the arm.
- \$TOL\_JNT: it has been replaced by two mono-dimensional arrays: \$TOL\_JNT\_COARSE and \$TOL\_JNT\_FINE.
- \$NUM\_SC\_BOARD has been renamed to \$NUM\_MCPS.
- \$A\_ALONG1 has been renamed in \$A\_ALONG\_1D
- \$A\_ALONG2 has been renamed in \$A\_ALONG\_2D
- \$C\_ALONG1 has been renamed in \$C\_ALONG\_1D
- \$C\_ALONG2 has been renamed in \$C\_ALONG\_2D
- \$D\_ALONG1 has been renamed in \$D\_ALONG\_2D
- \$D\_ALONG2 has been renamed in \$D\_ALONG\_2D
- \$S\_ALONG1 has been renamed in \$M\_ALONG\_1D

- \$S\_ALONG2 has been renamed in \$M\_ALONG\_2D
- \$A\_AREAL1 has been renamed in \$A\_AREAL\_1D
- \$A\_AREAL2 has been renamed in \$A\_AREAL\_2D
- \$C\_AREAL1 has been renamed in \$C\_REAL\_1D
- \$C\_AREAL2 has been renamed in \$C\_REAL\_2D
- \$D\_AREAL1 has been renamed in \$D\_AREAL\_2D
- \$D\_AREAL2 has been renamed in \$D\_AREAL\_2D
- \$S\_AREAL1 has been renamed in \$M\_AREAL\_1D
- \$S\_AREAL2 has been renamed in \$M\_AREAL\_2D

#### **14.6.4 \$FDIN, \$FDOUT, \$SDIN, \$SDOUT**

The meaning of some elements has been adapted to the architecture of the new C4G Controller. Please refer to [Chap. INPUT/OUTPUT Port Arrays](#) for the description of each element.

#### **14.6.5 BIT referenced System Variables**

Some bits of system variables do not have, on C4G Controller, the same meaning they used to have on C3G. Currently they are not used but, in the future, they will assume a specific meaning on C4G. PLD2 programs which were referencing to those bits on C3G should be modified because this kind of situation will not be detected by the translation of the program on C4G. Here below is reported a list of such bits:

- \$CNTRL\_INIT: bit 1, 3, 4, 5, 6, 12, 15, 16.
- \$CNTRL\_CNFG: bit 2, 6.
- \$PROG\_CNFG: bit 7
- \$RCVR\_TYPE: bit 9 has no longer a meaning in C4G Controller.
- \$RBT\_CNFG: bit 1-8, 12-21, 25-26, 28-30.
- \$SYS\_STATE: Bit 10, 12, 13, 19 and 22 are currently reserved.

#### **14.6.6 Array element referenced System Variables**

Some array elements of system variables do not have, on C4G Controller, the same meaning they used to have on C3G Controller. Currently they are not used but, in the future, they will assume a specific meaning on C4G Controller. PLD2 programs which were referencing to those elements on C3G Controller, should be modified because such a situation will not be detected by the program translation on C4G Controller. Here below a list of such elements is reported:

- \$TUNE elements 4,5,19, 22
- \$DFT\_SPD element 1

#### **14.6.7 Miscellaneous**

- \$RB\_FAMILY Only the robots supported by C4G Controller are handled. Please refer to [Chap. Predefined Variables List](#) for further details.

- \$CAL\_DATA has been changed from an ARRAY of INTEGER to an ARRAY of REAL.
- \$FOLL\_ERR has been changed to an ARRAY of REAL.
- \$NUM\_TIMERS The number of elements has been changed from 50 to 100.

## 14.7 Predefined constants

The following predefined constants have been removed:

- FLY\_HIGH,
- COM\_STOP1\_5,
- MMUX\_A, MMUX\_B, MMUX\_ON, MMUX\_OFF, MMUX\_SET.

The predefined constant EC\_USER has been replaced by EC\_USR1 and EC\_USR2.

## 14.8 Built-in routines and functions

A detailed description is supplied about the following topics:

- [Communication Device Control](#)
- [SYS\\_CALL](#)
- [WIN\\_LOAD](#)
- [Built-ins that have been removed](#)

### 14.8.1 Communication Device Control

- DV\_CTRL has been renamed in DV4\_CTRL. A cleaning operation has been applied to this function in order to remove what cannot be applied to the new Controller. Please refer to the description of [DV4\\_CTRL Built-In Procedure](#).  
Please note that COMP\_PORT\_GET and COM\_PORT\_SET instructions of C3G Controller have been moved in two different DV4\_CTRLs (with the operation code of number 3 and 4).
- DV\_STATE has been renamed in DV4\_state.

### 14.8.2 SYS\_CALL

The syntax of such a built-in procedure is still the same of C3G Controller. However, passing on C4G Controller, some commands have been removed, some other have been added and some more have been moved from a command branch to another.

It is recommended to check all sys\_calls that are written in the programs in order to determine:

- if the command sigla is still supported;
- if it has the same meaning.

Also a check on parameters and options is recommended.

Please refer to the **Use of C4G Control Unit** Manual, System Commands section, and to [SYS\\_CALL Built-In Procedure](#) description.

For file related commands, the directory specification and setting of the default directory should be considered.

### 14.8.3 WIN\_LOAD

Binary files containing windows previously created on C3G Controller by means of WIN\_CREATE and WIN\_SAVE procedures, cannot be directly loaded on C4G Controller. It is needed to re-create the windows on C4G Controller and issue WIN\_SAVE built-in procedure directly on C4G Controller.

### 14.8.4 Built-ins that have been removed

- AUX\_MMUX
- COM\_PORT\_SET2 and COM\_PORT\_GET. These are now part of DV4\_CNTRL.
- DV\_STATE. It has been renamed DV4\_STATE.
- SCRN\_FONT
- SYS\_VAR\_GET and SYS\_VAR\_SET
- CNTRL\_SET, CNTRL\_GET
- All PLC built-in routines, marked with the initial character ‘!’ have been removed.

## 14.9 Open File Statement

The OPEN FILE statement, when issued with the ‘rw’ specification, assumes that the file is already present in the directory otherwise an error is returned. This differs from C3G Controller, because in such a case an empty file was opened.

## 14.10 Conditions

Handling of conditions is the same as for C3G Controller. Some events have been added in the list of system events.

As far as concerns the class of user error events, the predefined constant EC\_USER has been replaced by EC\_USR1 and EC\_USR2. They identify the range of errors from 43008 to 43519 (EC\_USR1) and from 43520 to 44031 (EC\_USR2).

Please refer to [ERROR Events](#) section for further details.

Due to the fact that the value of some predefined constants changed, it is strongly recommended never to use the predefined value, but the identifier. For example, do not write WHEN ERROR CLASS 78, but WHEN ERROR CLASS EC\_TRAP.

## 14.11 Swapping to a foreign language

On C3G Controller, for changing the Controller language it was necessary to download a specific system file during the system download phase. On C4G Controller the language swapping is immediate and is obtained by issuing the command SetControllerLanguage<Language\_specification> or by pressing F1 key in the Home

Page of TP4i/WiTP Teach Pendant, and choosing the required language. It is not needed to restart the system. The effect is immediate. At the PDL2 level, changing of a language is detected by the system event WHEN EVENT 154.

## 14.12 System menu Commands

A certain amount of commands has been added and some others have been removed. About the added ones please consult the **Use of C4G Control Unit** Manual, System Commands section

One of the major differences from C3G Controller is that the Controller asks the user to Login in order to determine the access rights to be applied to the working session. The Login is done by means of the SetLogin command. The Login profile should be defined using the Configure Controller Login set of commands.

Detailed information is supplied about the following topics:

- [Removed System Menu Commands](#)
- [‘Configure’ commands](#)
- [‘Filer’ commands](#)
- [‘Input/Output’ commands](#)
- [Utility Comm Mount / Dism](#)
- [PLC commands](#)
- [Replaced Commands](#)

### 14.12.1 Removed System Menu Commands

The listed below commands have been removed. Therefore, the SYS\_CALL of these commands (indicated within round brackets) will go in error or will have a different meaning when performed on C4G Controller. Please refer to **Use of C4G Control Unit** Manual, for further details about the available commands.

### 14.12.2 ‘Configure’ commands

- Configure Controller Password and Set Controller Protect  
The protection level is now determined by the way the user logs in.
- ConfigureControllerRestartWarm  
On the C4G there is only one Restart modality.
- ConfigureControllerRestartBootmon.  
On the C4G the corresponding command is ConfigureControllerRestartMonitor
- Configure IO-Config  
The I/O configuration is done by the I/O configuration tool that runs on the PC.
- DisplayArmResolver /Hex  
This command originally was DisplayArmResolver. The hexadecimal visualization has been removed.

### 14.12.3 ‘Filer’ commands

- Filer Utility Fd\_formatHigh / Low
- FilerUtilityVerify

### 14.12.4 ‘Input/Output’ commands

- Set View\_io
- Set Update\_io
- Set TotalUnforce

### 14.12.5 Utility Comm Mount / Dism

The TISOFT, KERMIT and DDCMP are not supported anymore and the related submenus are not present.

### 14.12.6 PLC commands

- Display RII
- Memory RII Load e Save
- Program RII Activate, Clear\_Plc, Go, Restart, Single\_Scan, View

### 14.12.7 Replaced Commands

- Configure Load has now a submenu. Configure Load All and Configure Load Categories.
- Configure Save has now a submenu. Configure Save All and Configure Save Categories.
- Configure Controller Restart Reboot
  - It is now replaced by Configure Controller Restart Reload with the submenu for indicating the kind of load requested.
- Configure Controller Restart Bootmon
  - It is now replaces by Configure Controller Restart Monitor which activates the Cmon instead of the Dmon.
- Utility Communication Mount CrtEmu
  - It is now replaced by the Utility Communication Mount C4G\_Int which activates the communication protocol with the Winc4g via serial line, USB or network.

## 14.13 Compressed files

A compressed archive on the C4G is a .ZIP file. The .CPK are no more handled and the corresponding ZIP should be created before being copying the archive on C4G Controller.

## 14.14 Error and action Logging

On C3G Controller the latest \$NUM\_ELOG errors were stored in the ERROR.LOG file. It was possible to see the content of this ASCII file by printing it on the system scroll window or on a file.

On C4G Controller, errors are stored in a certain number (\$NUM\_ELOG\_FILES) of binary files (named <\$SYS\_ID><progressive\_number>.LBE) that reside in a specific directory (UD:\SYS\LOG). To view the lastest errors occurred in the system, there are several possibilities: either issue the UtilityLogError command; issue a FilerPrint of the file; issue a View Errors from WINC4G program that runs on the PC; or open the file from WINC4G file listing environment.

A new feature on C4G Controller is the logging of user actions (mainly commands issued and restart or power failure recovery actions). Like for errors, actions are stored in binary files in the directory UD:\SYS\LOG. The name is <\$SYS\_ID><progressive\_number>.LBA. The viewing mechanism are similar to what previously described for error.

## 14.15 PC interface

The PC interface to the C4G controller is the WINC4G program. It is the correspondent to the WINPCINT/PCINT of C3G Controller Unit but with much more features. Please refer to **Use of C4G Control Unit Manual**, for further details.

## 14.16 Fieldbuses configuration

The way of configuring fieldbuses changed (see system variables with prefix 'FB\_'). Please refer to the **Communications** Manual for further details.

## 14.17 MMUX (motor multiplexer)

This feature does not exist anymore because 10 axes are handled. All related built-in routines, predefined constants and system variables have been removed.

# 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 TP4i/WiTP Teach Pendant.

Character values enclosed in parentheses ( ) are ASCII control characters, which are displayed as graphics characters when printed to the screen of TP4i/WiTP Teach Pendant.



**Note that, on a PC (when WINC4G 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').

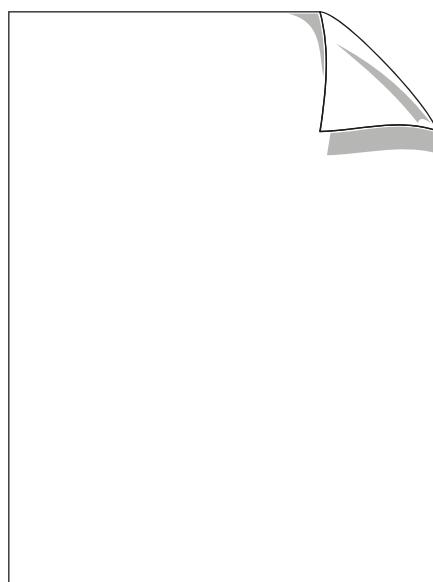
## 15.1 Characters Table

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	<b>K</b>	101	65	e	<b>e</b>	127	7F	(DEL)	<b>Δ</b>
076	4C	L	<b>L</b>	102	66	f	<b>f</b>	128	80		<b>Ç</b>
077	4D	M	<b>M</b>	103	67	g	<b>g</b>	129	81		<b>ü</b>
078	4E	N	<b>N</b>	104	68	h	<b>h</b>	130	82		<b>é</b>
079	4F	O	<b>O</b>	105	69	i	<b>i</b>	131	83		<b>â</b>
080	50	P	<b>P</b>	106	6A	j	<b>j</b>	132	84		<b>à</b>
081	51	Q	<b>Q</b>	107	6B	k	<b>k</b>	133	85		<b>å</b>
082	52	R	<b>R</b>	108	6C	l	<b>l</b>	134	86		<b>ç</b>
083	53	S	<b>S</b>	109	6D	m	<b>m</b>	135	87		<b>ê</b>
084	54	T	<b>T</b>	110	6E	n	<b>n</b>	136	88		<b>ë</b>
085	55	U	<b>U</b>	111	6F	o	<b>o</b>	137	89		<b>î</b>
086	56	V	<b>V</b>	112	70	p	<b>p</b>	138	8A		<b>ï</b>
087	57	W	<b>W</b>	113	71	q	<b>q</b>	139	8B		<b>ë</b>
088	58	X	<b>X</b>	114	72	r	<b>r</b>	140	8C		<b>í</b>
089	59	Y	<b>Y</b>	115	73	s	<b>s</b>	141	8D		<b>î</b>
090	5A	Z	<b>Z</b>	116	74	t	<b>t</b>	142	8E		<b>ä</b>
091	5B	[	<b>[</b>	117	75	u	<b>u</b>	143	8F		<b>ä</b>
092	5C	\	<b>\</b>	118	76	v	<b>v</b>	144	90		<b>é</b>
093	5D	]	<b>]</b>	119	77	w	<b>w</b>	145	91		<b>æ</b>
094	5E	^	<b>^</b>	120	78	x	<b>x</b>	146	92		<b>æ</b>
095	5F	-	<b>-</b>	121	79	y	<b>y</b>	147	93		<b>ö</b>
096	60	'	<b>'</b>	122	7A	z	<b>z</b>	148	94		<b>ö</b>
097	61	a	<b>a</b>	123	7B	{	<b>{</b>	149	95		<b>ö</b>
098	62	b	<b>b</b>	124	7C		<b> </b>	150	96		<b>ö</b>
099	63	c	<b>c</b>	125	7D	}	<b>}</b>	151	97		<b>û</b>
100	64	d	<b>d</b>	126	7E	~	<b>~</b>	152	98		<b>ÿ</b>



Dec	Hex	Value	Char
231	E7		⌚
232	E8		∅
233	E9		♾
234	EA		♾
235	EB		δ
236	EC		♾
237	ED		∅
238	EE		€
239	EF		∩
240	F0		≡
241	F1		±
242	F2		Σ
243	F3		≤
244	F4		≥
245	F5		∫
246	F6		÷
247	F7		≈
248	F8		°
249	F9		•
250	FA		•
251	FB		√
252	FC		¤
253	FD		²
254	FE		█
255	FF		



# 16. APPENDIX B - CUSTOMIZATIONS ON THE TP

## 16.1 Introduction

Current chapter contains information about the following topics:

- [User table creation from DATA environment](#)
- [Handling TP4i/WiTP right menu](#)
- [Customizing the PDL2 Statements insertion, in IDE environment.](#)

## 16.2 User table creation from DATA environment



Please, refer to Use of C4G manual in DATA Page section, for further information.

The PDL2 language allows the user to create a table and add it to the TP4i/WiTP interface in the DATA page.

The operation to be done are as follows:

- a. Table type definition: a RECORD must be defined which fields are the columns of the table.
- b. Table elements definition: the rows of the table are variables declared of the above type.
- c. Table viewing: the [TABLE\\_ADD Built-In Procedure](#) must be called.

Here follows a sample program.

```
PROGRAM exampletable NOHOLD
TYPE usertable = RECORD
    col1 : INTEGER
    col2 : BOOLEAN
    col3 : REAL
ENDRECORD

VAR line1, line2 : usertable
    line3_6 : ARRAY[4] OF usertable
BEGIN
    line1.col1:=10
    line1.col2:=TRUE
    line1.col3 :=4.5
```

```
TABLE_ADD( 'userTable' , $PROG_NAME , 'TabUser' )
```

```
END exampletable
```

- The table contents are stored in a .VAR file specified in the [TABLE\\_ADD Built-In Procedure](#).
- The table is removed from the DATA Page issuing the [TABLE\\_DEL Built-In Procedure](#).

In the following paragraphs detailed information is given about:

- [Properties](#)
- [Variable <type>\\_signal](#)
- [Example program for a table creation.](#)

## 16.2.1 Properties

It is possible to associate some properties [to the whole table](#) or [to fields](#) of it. Detailed information is provided in the following paragraphs.

For doing it, some variables have to be defined in the program header, according to the following syntax:

**<type\_name>\_<operation>** : when the operation is applied to the table.

**<type\_name>\_<field\_name>\_<operation>** : when the operation is applied to a field of the table.

**<type\_name>\_<field\_name>\_<field\_of\_a\_position>\_<operation>** : when the operation is applied to a field of a positional variable belonging to the table.

**<type\_name>** is the table;

**<field\_name>** is the field of the table;

**<field\_of\_a\_position>** is a field of a positional variable belonging to the table;

**<operation>** is one of the predefined functions listed here below.

They are grouped as follows:

- [Table \(global\) Properties](#)
- [Field Properties](#)
- [Field of a POSITION Properties.](#)

### 16.2.1.1 Table (global) Properties

The listed below properties apply to the whole table (i.e. they are “global” to the table).



**NOTE that the Table Property Name must always be preceded by <type\_name> (i.e. the table name).**

**Example:**

**if the table name is “table”, the “[\\_get\\_title](#)” property must be written as**  

table_get_title
-----------------

- [\\_get\\_active\\_element](#)
- [\\_get\\_auto\\_editable](#)

- [\\_get\\_can\\_auto\\_apply](#)
- [\\_get\\_max\\_entries](#)
- [\\_get\\_msg\\_on\\_save](#)
- [\\_get\\_title](#)
- [\\_get\\_var\\_path](#)
- [\\_integer\\_get\\_max](#)
- [\\_integer\\_get\\_min](#)
- [\\_integer\\_get\\_step](#)
- [\\_real\\_get\\_max](#)
- [\\_real\\_get\\_min](#)
- [\\_real\\_get\\_step.](#)

#### 16.2.1.1.1    [get active element](#)

<i>Name</i>	<a href="#"><u>_get_active_element</u></a>
<i>Description</i>	Get the active element in the table
<i>Data type</i>	STRING
<i>Read upon</i>	modifying the value of the variable (it is used for monitoring a variable)
<i>Default value</i>	none

#### 16.2.1.1.2    [get auto editable](#)

<i>Name</i>	<a href="#"><u>_get_auto_editable</u></a>
<i>Description</i>	Enabling flag for the table modification when the state of the controller is LOCAL. It is checked before enabling the modification of a field.
<i>Data type</i>	BOOLEAN
<i>Read upon</i>	table modification (of a field)
<i>Default value</i>	modifying is not enabled in LOCAL state

#### 16.2.1.1.3    [get can auto apply](#)

<i>Name</i>	<a href="#"><u>_get_can_auto_apply</u></a>
<i>Description</i>	Enabling flag for the automatic apply function, which consists in the copy of the data viewed, in the table, directly in variables loaded in execution memory.
<i>Data type</i>	BOOLEAN
<i>Read upon</i>	table opening
<i>Default value</i>	Auto apply is enabled by default

16.2.1.1.4 get max entries

Name	_get_max_entries
Description	Maximum number of rows in the table
Data type	INTEGER
Read upon	adding a new row to the table
Default value	none

16.2.1.1.5 get msg on save

Name	_get_msg_on_save
Description	Generation of a message upon the event of table saving
Data type	STRING
Read upon	table saving
Default value	none

16.2.1.1.6 get title

Name	_get_title
Description	Table name
Data type	STRING
Read upon	table opening
Default value	table name

16.2.1.1.7 get var path

Name	_get_var_path
Description	Path (directory and file name) in which saving the .VAR file containing the table data. The '\' character used in the directory path specification must be replaced by '\\'
Data type	STRING
Read upon	table saving
Default value	the same path used upon table load

16.2.1.1.8 integer get max

Name	_integer_get_max
Description	Maximum value for INTEGER fields
Data type	INTEGER
Read upon	table opening

<i>Default value</i>	none
----------------------	------

#### 16.2.1.1.9    *integer\_get\_min*

<i>Name</i>	_integer_get_min
<i>Description</i>	Minimum value for INTEGER fields
<i>Data type</i>	INTEGER
<i>Read upon</i>	table opening
<i>Default value</i>	none

#### 16.2.1.1.10    *integer\_get\_step*

<i>Name</i>	_integer_get_step
<i>Description</i>	Incremental and decremental step for INTEGER fields
<i>Data type</i>	INTEGER
<i>Read upon</i>	table opening
<i>Default value</i>	1

#### 16.2.1.1.11    *real\_get\_max*

<i>Name</i>	_real_get_max
<i>Description</i>	Maximum value for REAL fields
<i>Data type</i>	REAL
<i>Read upon</i>	table opening
<i>Default value</i>	none

#### 16.2.1.1.12    *real\_get\_min*

<i>Name</i>	_real_get_min
<i>Description</i>	Minimum value for REAL fields
<i>Data type</i>	REAL
<i>Read upon</i>	table opening
<i>Default value</i>	none

#### 16.2.1.1.13    *real\_get\_step*

<i>Name</i>	_real_get_step
<i>Description</i>	Incremental and decremental step for REAL fields

<i>Data type</i>	REAL
<i>Read upon</i>	table opening
<i>Default value</i>	0.1

### 16.2.1.2 Field Properties

The listed below properties apply to the fields of the table.



**NOTE** that the Field Property Name must always be preceded by both <type\_name> and <field\_name> (i.e. the table name and name of the field which it applies to), separated by the ‘\_’ character.

**Example:**

if the table name is “table”, and the field name is “curr”, the “\_get\_title” field property must be written as

table\_curr\_get\_title

- \_get\_editable
- \_get\_item\_list
- \_get\_item\_values
- \_get\_max
- \_get\_min
- \_get\_msg\_on\_change
- \_get\_step
- \_get\_tip
- \_get\_title
- \_get\_visible.

#### 16.2.1.2.1 \_get\_editable

<i>Name</i>	<u>_get_editable</u>
<i>Description</i>	Read-only flag for the field
<i>Data type</i>	BOOLEAN
<i>Read upon</i>	enabling the field modification
<i>Default value</i>	FALSE, which means that the field is enabled to the modification

#### 16.2.1.2.2 \_get\_item\_list

<i>Name</i>	<u>_get_item_list</u>
<i>Description</i>	List of possible items associated to a field of the table
<i>Data type</i>	ARRAY OF STRING
<i>Read upon</i>	table opening
<i>Default value</i>	none

#### 16.2.1.2.3 get\_item\_values

Name	_get_item_values
Description	List of values related to each item listed with the <a href="#"><u>_get_item_list</u></a> property
Data type	ARRAY OF INTEGER
Read upon	table opening
Default value	0 .. [max_items-1]

#### 16.2.1.2.4 get\_max

Name	_get_max
Description	Maximum value for the field
Data type	INTEGER or REAL
Read upon	selecting the field for being modified
Default value	maximum limit of the PDL2 data type

#### 16.2.1.2.5 get\_min

Name	_get_min
Description	Minimum value for the field
Data type	INTEGER or REAL
Read upon	selecting the field for being modified
Default value	minimum limit of the PDL2 data type

#### 16.2.1.2.6 get\_msg\_on\_change

Name	_get_msg_on_change
Description	Prompt a message after a field modification
Data type	STRING
Read upon	Apply command is issued on the field
Default value	none

#### 16.2.1.2.7 get\_step

Name	_get_step
Description	Incremental/decremental step for the field
Data type	INTEGER or REAL
Read upon	selecting the field for being modified

<i>Default value</i>	1 for INTEGERS and 0.1 for REALs
----------------------	----------------------------------

#### 16.2.1.2.8 [get\\_tip](#)

<i>Name</i>	_get_tip
<i>Description</i>	Help string assignement
<i>Data type</i>	STRING
<i>Read upon</i>	table opening
<i>Default value</i>	no Help string

#### 16.2.1.2.9 [get\\_title](#)

<i>Name</i>	_get_title
<i>Description</i>	Field name
<i>Data type</i>	STRING
<i>Read upon</i>	table opening
<i>Default value</i>	field name

#### 16.2.1.2.10 [get\\_visible](#)

<i>Name</i>	_get_visible
<i>Description</i>	Flag for enabling the viewing of a field of the table
<i>Data type</i>	BOOLEAN
<i>Read upon</i>	table opening
<i>Default value</i>	TRUE, which means that the field is always viewed

### 16.2.1.3 Field of a POSITION Properties

The listed below properties apply to the Position type fields of a table.



**NOTE that the Positional Fields Property Name must always be preceded by <type\_name>, <field\_name> and <field\_of\_a\_position> (i.e. the table name, the name of the field and the name of the positional field, which it applies to), separated by the ‘\_’ character.**

**Example:**

**if the table name is “table”, the field name is “pos1” and the field of the position is “x”, the “[get\\_max](#)” field property must be written as**

`table_pos1_x_get_max`

- [get\\_max](#)
- [get\\_step](#)
- [get\\_min](#).

#### 16.2.1.3.1    *get\_max*

Name	_get_max
Description	Maximum value for a field of a POSITION
Data type	INTEGER or REAL
Read upon	selecting a field for being modified
Default value	max limit of the PDL2 data type

#### 16.2.1.3.2    *get\_step*

Name	_get_step
Description	Increment of a POSITION field
Data type	INTEGER or REAL
Read upon	selecting a field for being modified
Default value	1 for INTEGERS and 0.1 for REALs

#### 16.2.1.3.3    *get\_min*

Name	_get_min
Description	Minimum value for a field of a POSITION
Data type	INTEGER or REAL
Read upon	selecting a field for being modified
Default value	minimum limit of the PDL2 data type

## 16.2.2 Variable <type>\_signal

Such a variable is used by the user interface to signal an event to the PDL2 program which handles the tables. The PDL2 program has to check the variable value, in order to know whether the event occurred or not. The predefined value (0) indicates that none happened. In case of special events (for example table saving), for which a notification to the PDL2 program is needed, the user interface modifies its value as follows:

Event type	Variable value	When is it modified?
Saving a table	1	At the end of saving operations, if successfully completed
Reloading a table	2	At the end of reloading operations, if successfully completed

It is up to the PDL2 program to reset the variable (to the default value of 0) at the end of the event service routine, otherwise a further event of the same type could be not detected (since the value of the variable wouldn't change!).

If such a variable is not in the table, the user interface doesn't handle it at all.

### 16.2.3 Example program for a table creation

```

PROGRAM test_tab NOHOLD
TYPE tab_def = RECORD
  curr : REAL
  curr1 : REAL
  volt1 : REAL
  volt2 : REAL
  pressure : REAL
  spd : INTEGER
  acc : INTEGER
  jerk : INTEGER
  dec : INTEGER
  mass : REAL
  enabled : BOOLEAN
  stokeend : BOOLEAN
  opened : BOOLEAN
  pos : POSITION
ENDRECORD

VAR
  tab_def_get_title : STRING[13] ('Tabella test ') NOSAVE
  tab_def_max_entries : INTEGER (5)
  tab_def_get_msg_on_save : STRING[44] ('Messaggio di prova per
    salvataggio') NOSAVE
  tab_def_curr_get_title : STRING[7] ('Current') NOSAVE
  tab_def_curr_get_tip : STRING[13] ('Total current') NOSAVE
  tab_def_curr_get_max_value : REAL (3.0)
  tab_def_curr_get_min_value : REAL (-3.0)
  tab_def_curr1_get_visible : BOOLEAN (FALSE)
  tab_def_get_auto_editable : BOOLEAN (FALSE)
  tab_def_spd_get_item_list : ARRAY[3] OF STRING[10]
  tab_def_mass_get_editable : BOOLEAN (TRUE)
  tab_def_mass_get_msg_on_change : STRING[30] ('Messaggio di
    modifica di prova') NOSAVE
  tab_def_curr_get_editable : BOOLEAN (TRUE)

  tab1 : tab_def
  tab2 : tab_def
  tab3 : tab_def
  tab4 : tab_def
  tab5 : tab_def
  tab6 : tab_def
  tab7 : ARRAY[3] OF tab_def

BEGIN
  TABLE_ADD('tab_def', $PROG_NAME)

  tab_def_spd_get_item_list[1] := 'low'
  tab_def_spd_get_item_list[2] := 'medium'
  tab_def_spd_get_item_list[3] := 'high'

END test_tab

```

## 16.3 Handling TP4i/WiTP right menu



**Please, refer to Use of C4G manual in Chap.6 - Use of the Teach Pendant, DISPLAY section, for further information.**

There are 6 keys on the TP4i/WiTP that can be programmed by the user basing on the application to be developed.

Each page of the application could need more than 6 keys; pressing MORE key (on the right of the right menu) the remaining softkeys of the current page are displayed.

The changing from one page to another is circularly handled via the MORE key and the swapping occurs when the last softkey of the page has been displayed.

It is possible to disable a softkey but it is not allowed to disable a page of softkeys.

Each softkey can be in the following status:

- Enabled or disabled.
- The pressure of a disabled softkey does not have any effect. Please note that the key aspect remains unchanged in respect to when it is enabled: the PDL2 softkey handler program has to modify the key in terms of color, icon, text so to underline the disabled key status.
- Pressed or Released
- When the softkey is in the pressed state, the softkey is represented like if it were embedded.

A configuration file in XML format has to be written containing the definition of the softkeys; this file must be stored in a predefined location of the file system. Please contact COMAU at service.robotics@comau.com for further details. The user should have the access rights for accessing to the LD: device.

Upon the Controller restart, all the files contained in the above directory are automatically loaded, in alphabetical order, on the TP4i/WiTP.

Each page corresponds to the definition of a group of softkeys, which can be more than 6. The syntax is as follows:

```

<?xml version="1.0"?>

<NewMenu>
    <group>
        <progName>GroupName</progName>
        <numItems>Number</numItems>
        <arName>Names</arName>
        <arIconName>Icons</arIconName>
        <arStatus>Colours</arStatus>
        <arEnabled>Enabled</arEnabled>
        <evPressed>NumberOfEvent</evPressed>
        <evLongPressed> NumberOfEvent </evLongPressed>
        <evReleased> NumberOfEvent </evReleased>
    </group>
</NewMenu>

```

The words included between < > should always be present. The words specified in bold have to be replaced by a true parameter.

Example: 2 XML files are defined. In the first file, one group with 8 softkeys is defined; in the second one, 2 groups with 3 softkeys each are defined. TP4i/WiTP will create the following pages:

- One page with the first 6 softkeys of the group of the first file
- One page with the remaining 2 softkeys of the group of the first file
- One page with the 3 softkeys of the first group of the second file
- One page with the 3 softkeys of the second group of the second file.

A detailed description follows about the following topics:

- [XML Tag Parameters](#)
- [Softkey Pressure and Release events](#)
- [Example of XML configuration file](#)
- [Example of right menu configuration](#)

### 16.3.1 XML Tag Parameters

Here follows the description of the XML tag parameters to be defined in the group specification.

**<progName> ... </progName> **Softkeys handler****

The information required for handling each key belonging to a group are stored in variables which should be loaded in execution memory (Load command). The definition and the initialisation of these variables should be included in this program. The program should be loaded by the controller startup program.

**< numItems >...</numItems > **Number of items to be considered****

This XML parameter indicates the number of elements which should be considered inside the array variables which contain the properties for each softkey.

The arrays could have a greater dimension but the number of elements taken into account is the one indicated by this parameter.

**<arName>...</arName> **Labels for the softkeys****

This is the name of a PDL2 variable, defined in the <progName> program, which contains the labels associated to the softkeys of the group. This variable should be defined as an ARRAY OF STRING. The maximum number of characters of each label depends on the dimension of the key and by the font used for the label. The strings which overflow the key dimension are truncated.

The TP4i/WiTP continuously monitors the content of the array, and if it changes the effect on the softkey is immediate.

**<arIconName>...</ arIconName > **Icons for the softkeys****

This is the name of a PDL2 variable, defined in the <progName> program, which contains the icons associated to the softkeys of the group. This variable should be defined as an ARRAY OF STRING. The content of each element is the name of the file (.GIF) containing the icon. This file should be stored in the same folder of the .XML file. For example, if the element 5 of the PDL2 variable <arlIconName> defined in <progName> program has the value 'TEST', the TP4i/WiTP shows the icon present in TEST.GIF.

The TP4i/WiTP continuously monitors the content of the array, and if it changes, the effect on the softkey is immediate.

**< arEnabled >...</ arEnabled >** **Softkey Enabling / disabling flag**  
 This is the name of a PDL2 variable, defined in the <progName> program, which contains the enabling/disabling flag for the softkeys of the group. This variable should be defined as an ARRAY OF BOOLEAN. The value FALSE disallowes the pressure of the corresponding softkey. Please note that the look and feel of the softkey is not changed at all by the TP4i/WiTP; the softkey handler program <progName> should eventually handle a change of aspect of the key.

The TP4i/WiTP continuously monitors the content of the array, allowing the content to be changed anytime and the effect on the softkey is immediate.

**< arStatus>...</ arStatus >** **Softkey colour**

This is the name of a PDL2 variable, defined in the <progName> program, which contains the colour for the softkeys of the group. This variable should be defined as an ARRAY OF INTEGER. The possible values are the predefined constants related to the colours : WIN\_BLUE, WIN\_RED, WIN\_... (please refer to the PDL2 manual for the list of predefined variables).

The TP4i/WiTP continuously monitors the content of the array, and if it changes the effect on the softkey is immediate.

### 16.3.2 Softkey Pressure and Release events

There are three kind of events related to the pressure of a softkey:

- Softkey Pressure Event ([\*\*< evPressed >...</ evPressed >\*\*](#))
- Softkey Release Event ([\*\*< evReleased >...</ evReleased >\*\*](#))
- Softkey 'long lasting' pressure Event ([\*\*< evLongPressed>...</ evLongPressed >\*\*](#))

They are described here below.

**< evPressed >...</ evPressed >**

An event of key pressure can be associated to each softkey.

A user event CONDITION (WHEN EVENT ... ) statement should monitor the triggering of these kind of events. The class of user events is identified in the range of errors included between 49152 and 50175. This XML tag should contain the event number associated to the first softkey of the group. The TP4i/WiTP will automatically assign the events for the other softkeys of the group in a sequential order. For example, defining **<evPressed >50100</ evPressed>** in a group of 6 softkeys, the event 50100 will be generated upon the key pressure of the first softkey of the group, the event 50105 will be generated upon the key pressure of the 6th softkey of this group.

**< evReleased >...</ evReleased >**

An event of key release can be associated to each softkey.

The way for using it is the same as for the key pressure described above. The only difference is in the XML tag definition.

**< evLongPressed>...</ evLongPressed >**

This event triggers when the softkey of the TP4i/WiTP is maintained pressed for more than 2 seconds.

The way for using it is the same as for the key pressure described above. The only difference is in the XML tag definition.

< evShow>...</ evShow > **Force the viewing of a softkey group**  
 It is possible to force the viewing of a group of softkeys. For doing this a user event must be defined (WHEN EVENT..) which will trigger upon a SIGNAL EVENT of the same number. The TP4i/WiTP shows the group when the event triggers.

### 16.3.3 Example of XML configuration file

This example file defines two groups of softkeys. The first group has 4 keys with the names defined in 'nomi', the icons are defined in 'icone', the enabling is stored in 'en' and the colours are defined in 'colori'. The handler program is 'group1'.

The second group defines 2 softkeys and the attributes are stored in the same variables. The handler program is 'group2'.

```

<?xml version="1.0"?>

<NewMenu>
<group>
  <progName>group1</progName>
  <numItems>4</numItems>
  <arName>nomi</arName>
  <arIconName>icone</arIconName>
  <arStatus>colori</arStatus>
  <arEnabled>en</arEnabled>
  <evPressed>50100</evPressed>
  <evLongPressed>50120</evLongPressed>
  <evReleased>50130</evReleased>
</group>

<group>
  <progName>group2</progName>
  <numItems>2</numItems>
  <arName>nomi</arName>
  <arIconName>icone</arIconName>
  <arStatus>colori</arStatus>
  <arEnabled>en</arEnabled>
  <evPressed>50150</evPressed>
  <evLongPressed>50160</evLongPressed>
  <evReleased>50170</evReleased>
</group>

</NewMenu>

```

This example program, assigns a colour to the key which has been pressed and disables the other softkey.

```

PROGRAM group1 NOHOLD
VAR
  nomi : ARRAY[5] OF STRING[10]
  icone: ARRAY[5] OF STRING[10]
  en : ARRAY[5] OF BOOLEAN
  colori : ARRAY[5] OF INTEGER
BEGIN

```

```

CONDITION[1] NODISABLE :
    WHEN EVENT 50100 DO -- press. of the 1st softkey of the group
        colori[1] := WIN_RED -- make RED the first softkey
        en[2] := FALSE -- disable the second softkey
    WHEN EVENT 50101 DO -- press. of the 2nd softkey of the group
        colori[2] := WIN_GREEN -- make GREEN the first softkey
        en[1] := FALSE -- disable the first softkey
    ENDCONDITION
    ENABLE CONDITION[1]
CYCLE
---
DELAY 1000
---
END group1

```

### 16.3.4 Example of right menu configuration

This XML file defines a group with 5 softkeys. The handler program is called 'rightmenu'. The names are stored in 'nomi', the icons in 'icone', the enabling in 'en' and the colours in 'colori'. The event for the pressure of the first softkey is 50120. The event for the long pressure of the first softkey is 50120. The first event for releasing of the first softkey is 50000.

```

<?xml version="1.0"?>

<RightMenu>

<group>
<progName>rightmenu</progName>
<numItems>5</numItems>
<arName>nomi</arName>
<arIconName>icone</arIconName>
<arStatus>colori</arStatus>
<arEnabled>en</arEnabled>
<evPressed>50100</evPressed>
<evLongPressed>50120</evLongPressed>
<evReleased>50130</evReleased>
<evShow>50000</evShow>
</group>

</RightMenu>

```

This program initializes the first four softkeys of the right menu.

```

PROGRAM rightmenu NOHOLD
VAR
    nomi : ARRAY[5] OF STRING[10]
    icone : ARRAY[5] OF STRING[10]
    en :      ARRAY[5] OF BOOLEAN
    colori : ARRAY[5] OF INTEGER
BEGIN

```

```

-- first softkey initialization
nomi[1] := 'S1'
icone[1] := 'i1'
en[1] := TRUE
colori[1] := WIN_BLUE

-- second softkey initialization
nomi[2] := 'S2'
icone[2] := 'i1'
en[2] := TRUE
colori[2] := WIN_RED

nomi[3] := 'S3'
icone[3] := 'i1'
en[3] := TRUE
colori[3] := WIN_YELLOW

nomi[4] := 'S4'
icone[4] := 'i1'
en[4] := TRUE
colori[4] := WIN_GREEN

CONDITION[1] NODISABLE :
  WHEN EVENT 50100 DO
    colori[1] := WIN_RED
    colori[2] := WIN_GREEN
  WHEN EVENT 50101 DO
    colori[1] := WIN_PINK
    colori[2] := WIN_RED
ENDCONDITION
ENABLE CONDITION[1]

CYCLE
  DELAY 1000
  icone[2] := 'i2'
  DELAY 1000
  icone[2] := 'i1'
END rightmenu

```

## 16.4 Customizing the PDL2 Statements insertion, in IDE environment

In IDE environment (on the Teach Pendant), the user is allowed both to customize the PDL2 menu, adding new instructions to it, and to create a virtual numeric keypad to quickly insert them into a PDL2 program.

Here follows a detailed description of these two functionalities:

- [Adding User Instructions to the PDL2 menu](#)
- [Creating a “virtual” Numeric Keypad to insert User Instructions](#)

### 16.4.1 Adding User Instructions to the PDL2 menu

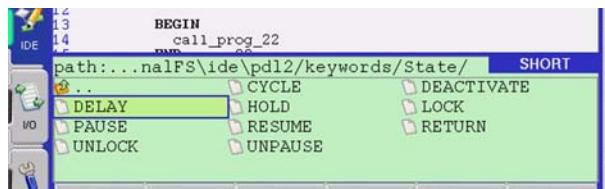
To do that, the user has to write an **xml** file to fully describe the structure of the being added instruction.



**To know how to quickly insert the added User Instruction, see next par. 16.4.2 Creating a “virtual” Numeric Keypad to insert User Instructions on page 16-20.**

The **xml** file must be included in the suitable directory of the Controller, so that it is listed together with all the already available instructions (see Fig. 16.1).

**Fig. 16.1 - Example of PDL2 instructions list, in IDE Page**



Three examples follow, of **xml** files, for

- [Adding a Statement,](#)
- [Adding a Routine,](#)
- [Adding a Built-in Routine.](#)

#### 16.4.1.1 Adding a Statement

In the following example, the **DELAY** instruction template is described by means of an **xml** file, to obtain the template shown in Fig. 16.2.

**Fig. 16.2 - DELAY statement template**

```
DELAY <time>
<keyword>
    <text>DELAY % </text>
    <param>
        <text>time</text>
        <keymode>num</keymode>
    </param>
</keyword>
```

- **<text>DELAY % </text>** states that a string will be displayed containing the instruction name (**DELAY**), with **one parameter**.
- The section included between **<param>** and **</param>** describes the structure of parameter:
  - the field name displayed in the template (**<text>time</text>**),
  - the alphanumeric keypad modality (**<keymode>num</keymode>**), associated to such a parameter.

### 16.4.1.2 Adding a Routine

In the following example, a Routine Calling template is described by means of an **xml** file, to obtain the template shown in Fig. 16.3.

**Fig. 16.3 - Routine Calling template**

```
A_TOOL(<index>)

<keyword>
  <import>
    <prog_name>tt_tool</prog_name>
    <type>
      <type_name>routines</type_name>
      <name>a_tool</name>
    </type>
  </import>

  <text>A_TOOL (%) </text>
  <param>
    <text>index</text>
    <keymode>num</keymode>
  </param>
</keyword>
```

- the section between **<import>** and **</import>**, describes any information which is needed to EXPORT the routine FROM
- **<prog\_name>tt\_tool</prog\_name>** is the name of the owning program (tt\_tool) the routine is to be EXPORTED FROM
- **<type\_name>routines</type\_name>** indicates the type of the object to be EXPORTED FROM
- **<name>a\_tool</name>** is the routine name
- **<text>A\_TOOL (%) </text>** states that a string will be displayed in the template, containing the routine name (**A\_TOOL**), with **one parameter**
- The section included between **<param>** and **</param>** describes the structure of parameter:
  - the field name displayed in the template (**<text>index</text>**),
  - the alphanumeric keypad modality (**<keymode>num</keymode>**), associated to such a parameter.

### 16.4.1.3 Adding a Built-in Routine

In the following example, the AUX\_COOP Built-in Routine calling template is described by means of an **xml** file, to obtain the template shown in Fig. 16.4.

**Fig. 16.4 - Built-in calling template**

```
AUX_COOP(<flag>, <aux_joint>, <arm_num>)

<keyword>
  <text>AUX_COOP (%, %, %) </text>
```

```

<param>
    <text>flag</text>
    <keymode>alpha</keymode>

    <context>ON</context>
    <context>OFF</context>
</param>

<param>
    <text>aux_joint</text>
    <keymode>num</keymode>
</param>

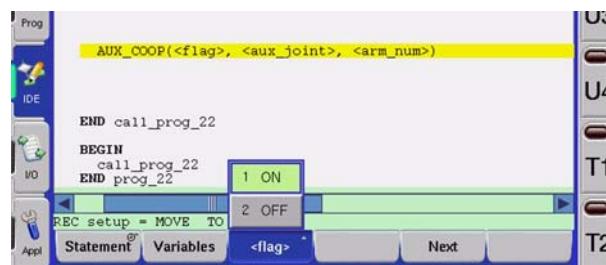
<param>
    <text>arm_num</text>
    <keymode>num</keymode>
</param>

</keyword>

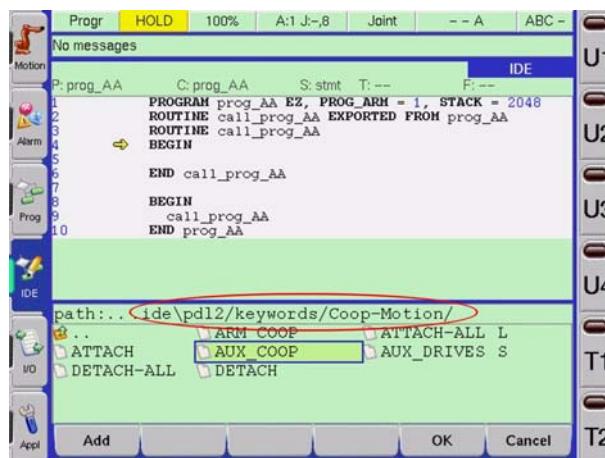
```

- **<text>AUX\_COOP (% , %, %) </text>** states that a string will be displayed in the template, containing the routine name (**AUX\_COOP**), with **three parameters**
- The three sections included between **<param>** and **</param>** describe the structure of the parameters:
  - the field names displayed in the template (**<text>flag</text>**, **<text>aux\_joint</text>**, **<text>arm\_num</text>**)
  - the alphanumeric keypad modality (**<keymode>num</keymode>**, **<keymode>alpha</keymode>**), associated to such parameters.
  - **<context>ON</context>** and **<context>OFF</context>** indicate the strings to be displayed in the menu which becomes available when the ‘flag’ field is selected (context key **F3** - see [Fig. 16.5](#)).

**Fig. 16.5 - ‘flag’ context menu**



When the **AUX\_COOP.xml** file is created in the suitable directory, the AUX\_COOP built-in routine is immediately available, as shown in [Fig. 16.6](#).

**Fig. 16.6 - Example - AUX\_COOP**

## 16.4.2 Creating a “virtual” Numeric Keypad to insert User Instructions

It is possible to create a Numeric Keypad to be displayed in the IDE Page, in order to quickly insert the **User Instructions** in PDL2 programs.



To know how to add a User Instruction to the PDL2 menu, see previous [par. 16.4.1 Adding User Instructions to the PDL2 menu on page 16-17](#).

To do that, it is needed to write an **xml** file as explained below.

An example follows about creating a Numeric Keypad (see [Fig. 16.7](#)) related to the Cooperative Motion built-ins.

```

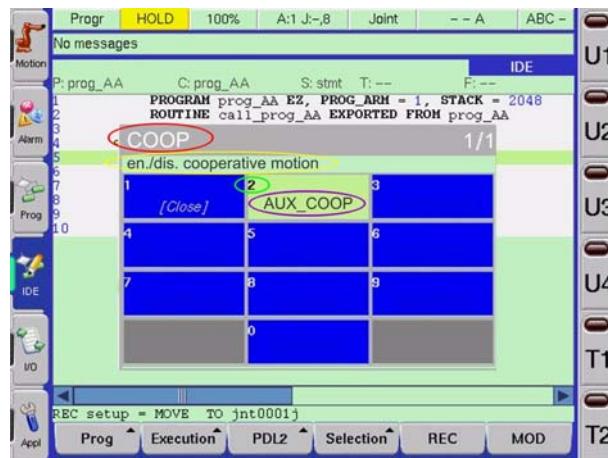
<numpad>
  <header>
    <title>COOP</title>
  </header>

  <shortcut>
    <numkey>2</numkey>
    <statement>AUX_COOP</statement>
    <help>
      <it>ab./disab. movimento cooperativo</it>
      <en>en./dis. cooperative motion</en>
    </help>
    <path>IDE\PDL2\keywords\Coop-Motion\AUX_COOP.xml</path>
  </shortcut>
</numpad>
```

- **<title>COOP</title>** states that the string ‘COOP’ will be displayed as the title of the table (red highlighted string in [Fig. 16.7](#));
- **<numkey>2</numkey>** states that the numeric key which is involved in the current shortcut definition, is key no.’2’ (green highlighted in [Fig. 16.7](#));

- **<statement>AUX\_COOP</statements>** states that the instruction associated to the numeric key (included in the **<numkey>** section) is called 'AUX\_COOP' (purple highlighted in Fig. 16.7);
- **<help>**  
**<it>**ab./disab. movimento cooperativo**</it>**  
**<en>**en./dis. cooperative motion**</en>**  
**</help>**  
 is the section which includes the help strings, in different languages. In the example shown in Fig. 16.7, the chosen help string is in English (yellow highlighted);
- **<path>IDE\PDL2\keywords\Coop-Motion\AUX\_COOP.xml</path>** indicates
  - path (LD:\TP4\TP4i\InternalFS\IDE\PDL2\keywords\Coop-Motion\ - see the red highlighted string in Fig. 16.6) and
  - name (AUX\_COOP.xml)
 to reach the file which has already been created to insert the AUX\_COOP icon in the PDL2 menu (see par. 16.4.1.3 Adding a Built-in Routine on page 16-18).

**Fig. 16.7 - Numeric Keypad**

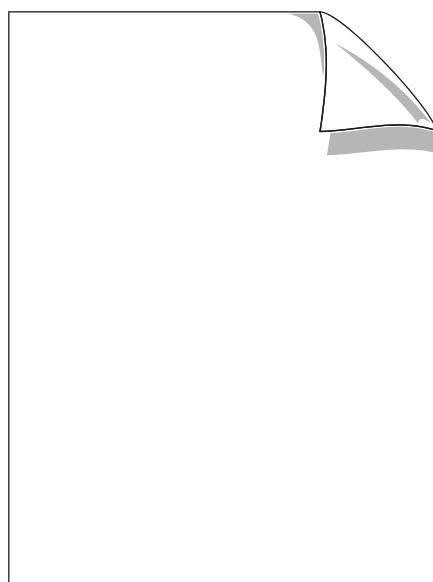


When a Numeric Keypad does exist, including a specified instruction, pressing **SHIFT+1** opens such a Numeric Keypad.

As soon as the numeric key (in our example is 2) is pressed, the corresponding template is inserted in the PDL2 program (in our example the one shown in Fig. 16.4), ready to be edited.



For further information, refer to [Use of C4G Unit manual, IDE Page, par.6.11.4 Numeric Keypad](#).



# 17. APPENDIX C - E-MAIL FUNCTIONALITY

## 17.1 Introduction

Current chapter describes the e-mail functionality which allows to send/receive e-mails by means of the C4G 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 C4G Controller
- Sending PDL2 commands via e-mail

## 17.2 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. 17.1 - \$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. 17.2 - \$EMAIL\_INT**

[1]: flags
<ul style="list-style-type: none"> <li>• 0x01: Feature is enabled</li> <li>• 0x04: use APOP authentication</li> <li>• 0x10: Use of memory or file system for incoming mail</li> <li>• 0x20: Do not remove the directory where attachments are stored upon system startup</li> </ul>
[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.

## 17.3 Sending/receiving e-mails on C4G Controller

A PDL2 program called “email” is shown below ([“email” program](#)): it allows to send and receive e-mails on C4G Controller.

[DV4\\_CNTRL Built-In Procedure](#) is to be used to handle such functionalities.



See [DV4\\_CNTRL Built-In Procedure](#) in [Chap. BUILT-IN Routines list](#) section for further information about the e-mail functionality parameters.

### 17.3.1 “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
    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])
      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

  DV4_CNTRL(ki_email_cfg, si_handle, ws_cfg, wi_cfg)

  WHILE vi_ext = 0 DO
    IF vi_first = 1 THEN
      vi_first := 0
      pu_help
    ENDIF
  ENDWHILE
END
  
```

```

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
            DV4_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)
        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)
            DV4_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
            DV4_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
            DV4_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
            DV4_CNTRL(ki_email_del, (si_handle), 1)
            WRITE LUN_CRT (*Delete OK.', NL)
        CASE (6): -- Help
            pu_help
        CASE (7): -- Exit
            DV4_CNTRL(ki_email_close, si_handle)
            WRITE LUN_CRT (*Program exit.', NL)
            vi_ext := 1
        ELSE:
            pu_help
        ENDSELECT
    ELSE
    ENDIF
ENDWHILE
END email

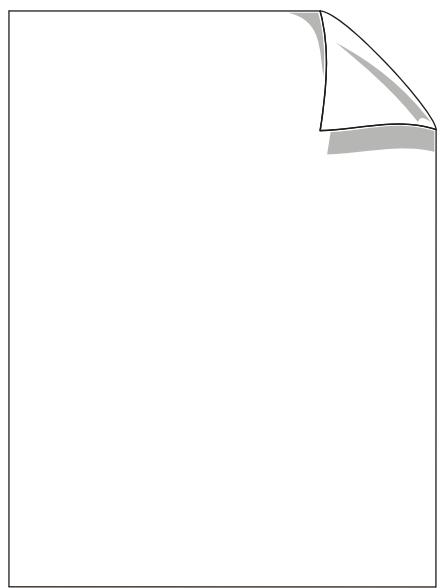
```

## 17.4 Sending PDL2 commands via e-mail

The user is allowed to send PDL2 commands to the C4G 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 *c4gmp* 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 *c4gmp* program is ok within a time interval of half an hour, more or less, since the generation time.





#### **Comau in the World**

##### **COMAU S.p.A.**

###### **Headquarters**

Via Rivalta, 30  
10095 Grugliasco -TO (Italy)  
Tel. +39-011-0049111  
Telefax +39-011-789356

##### **Powertrain Systems**

Via Rivalta 49  
10095 Grugliasco -TO (Italy)  
Tel. +39-011-0049111  
Telefax +39-011-0049701

##### **Body Welding & Assembly**

Strada Borgarett 22  
10040 Borgarett di Beinasco -TO (Italy)  
Tel. +39-011-0049111  
Telefax +39-011-3582705

##### **Robotics & Service**

Via Rivalta, 30  
10095 Grugliasco -TO (Italy)  
Tel. +39-011-0049111  
Telefax +39-011-0045481

##### **Engineering, Injection Moulds & Dies**

Via Bistagno 10  
10136 Torino (Italy)  
Tel. +39-011-00517111  
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**

Hanns-Klemm-Strasse 5  
D-71034 Böblingen (Germany)  
Tel. +49-7031-73400  
Telefax +49-7031-7340299

##### **Mecaner S.A.**

Calle Aita Gotzon 37  
48610 Urduliz - Vizcaya (Spain)  
Tel. +34-94-6769100  
Telefax +34-94-6769132

##### **Comau Sverige AB**

Kardanvangen 37  
SE 461  
38 Trollhattan (Sweden)  
Tel. +46-520-279730  
Telefax +46-520-279799

##### **Comau Poland Sp.Z.O.O.**

Ul. Turynska 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**

Panayir Mah. Buttumis\_Merkezi  
C Block Kat 5 no.1494  
16250 Osmangazi/Bursa (Turkey)  
Tel. +90-0-2242112873  
Telefax +90-0-2242112834

##### **Comau Pico 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, Tepotzotlan (Mexico)  
Tel. +52-5 8760644  
Telefax +52-5 8761837

##### **Comau Pico of 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 - São Paulo - SP - Brasil  
Cep. 04103-000  
Tel. +55-31-21236306  
Telefax +55-31-21233349

##### **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) Ltd.**

Hendrik van Eck Drive  
Riverside Industrial Area  
Uitenhage 6229 (South Africa)  
Tel. +27-41-9953600  
Telefax +27-41-9229652

##### **Comau SA Press Tool & Parts (Pty) Ltd.**

87 Baird Street  
Uitenhage 6229 (South Africa)  
Tel. +27-41-9953700  
Telefax +27-41-9924148

##### **Comau Korea**

40-9, Ugnam-Dong  
Changwon, Kyungnam (Korea)  
Tel. +82-55-2757015  
Telefax +82-55-2757016

##### **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 +86-21-68139621

##### **Comau India Pvt.Ltd.**

26/A/1, Kondhwa Budruk  
Pune Pin - 411048  
Maharashtra (India)  
Tel. +91-20-56780369  
Telefax +91-20-56002865

#### **COMAU Robotics services**

Repair: [repairs.robots@comau.com](mailto:repairs.robots@comau.com)

Training: [training.robots@comau.com](mailto:training.robots@comau.com)

Spare parts: [spares.robots@comau.com](mailto:spares.robots@comau.com)

Technical service: [service.robots@comau.com](mailto:service.robots@comau.com)

[comau.com/robotics](http://comau.com/robotics)