

# **FANUC Robot series**

**SYSTEM R-30iA/Mate, R-30iB/  
Mate, and R-30iB Plus/Mate Plus/  
Mini Plus/Compact Plus  
CONTROLLER**  
**7.50 and later**

**KAREL REFERENCE MANUAL**

**MARRC75KR07091E Rev N**

EFFMAN  
QUIRIONR

# ORIGINAL INSTRUCTIONS

Thank you very much for purchasing a FANUC robot.

Before using the robot, be sure to read the, *FANUC Robot series SAFETY HANDBOOK (B-80687EN)* and understand its contents.

- No part of this manual may be reproduced, copied, downloaded, translated into another language, published in any physical or electronic format, or in any other form, including the internet, or transmitted in whole or in part in any way without the prior written consent of FANUC or FANUC America Corporation.
- The appearance and specifications of this product are subject to change without notice.

The products in this manual are controlled based on Japan's "Foreign Exchange and Foreign Trade Law". The export from Japan may be subject to an export license by the government of Japan. Further, re-export to another country may be subject to the license of the government of the country from where the product is re-exported. Furthermore, the product may also be controlled by re-export regulations of the United States government. Should you wish to export or re-export these products, please contact FANUC for advice.

In this manual, we endeavor to include all pertinent matters. There are, however, a very large number of operations that must not or cannot be performed. Please assume that any operations that are not explicitly described as being possible are not possible.

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# SAFETY PRECAUTIONS

---

This chapter describes the precautions which must be followed to enable the safe use of the robot. Before using the robot, be sure to read this chapter thoroughly.

For detailed functions of the robot operation, read the relevant operator's manual to understand fully its specification.

For the safety of the operator and the system, follow all safety precautions when operating a robot and its peripheral equipment installed in a work cell. For safe use of FANUC robots, you must read and follow the instructions in the *FANUC Robot series SAFETY HANDBOOK (B-80687EN)*.

## PERSONNEL

---

Personnel can be classified as follows:

Operator:

- Turns the robot controller power ON/OFF
- Starts the robot program from the operator panel

Programmer or Teaching operator:

- Operates the robot
- Teaches the robot inside the safeguarded space

Maintenance technician:

- Operates the robot
- Teaches the robot inside the safeguarded space
- Performs maintenance (repair, adjustment, replacement)
- The operator is not allowed to work in the safeguarded space.
- The programmer or teaching operator and maintenance technician are allowed to work in the safeguarded space. Work carried out in the safeguarded space include transportation, installation, teaching, adjustment, and maintenance.
- To work inside the safeguarded space, the person must be trained on proper robot operation.

**Table s-1** lists the work outside the safeguarded space. In this table, the symbol “○” means the work is allowed to be carried out by the specified personnel.

**Table s-1 Work Performed Outside the Safeguarded Space**

	Operator	Programmer or Teaching Operator	Maintenance Technician
Turn power ON/OFF to Robot controller	○	○	○
Select operating mode (AUTO, T1, T2)		○	○
Select remote/local mode		○	○
Select robot program with teach pendant		○	○
Select robot program with external device		○	○
Start robot program with operator's panel	○	○	○

	Operator	Programmer or Teaching Operator	Maintenance Technician
Start robot program with teach pendant		<input type="radio"/>	<input type="radio"/>
Reset alarm with operator's panel		<input type="radio"/>	<input type="radio"/>
Reset alarm with teach pendant		<input type="radio"/>	<input type="radio"/>
Set data on teach pendant		<input type="radio"/>	<input type="radio"/>
Teaching with teach pendant		<input type="radio"/>	<input type="radio"/>
Emergency stop with operator's panel	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Emergency stop with teach pendant	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Operator panel maintenance			<input type="radio"/>
Teach pendant maintenance			<input type="radio"/>

During robot operation, programming and maintenance, the operator, programmer, teaching operator and maintenance engineer take care of their own safety using at least the following safety protectors:

- Use clothes, uniform, overall adequate for the work
- Safety shoes
- Helmet

## DEFINITION OF SAFETY NOTATIONS

To ensure the safety of users and prevent damage to the machine, this manual indicates each precaution on safety with "WARNING" or "CAUTION" according to its severity. Supplementary information is indicated by "NOTE". Read the contents of each "WARNING", "CAUTION" and "NOTE" before using the robot.

Symbol	Definitions
 <b>WARNING</b>	Used if hazard resulting in the death or serious injury of the user will be expected to occur if he or she fails to follow the approved procedure.
 <b>CAUTION</b>	Used if a hazard resulting in the minor or moderate injury of the user, or equipment damage may be expected to occur if he or she fails to follow the approved procedure.
<b>NOTE</b>	Used if a supplementary explanation not related to any of WARNING and CAUTION is to be indicated.

# TABLE OF CONTENTS

---



---

<b>SAFETY PRECAUTIONS.....</b>	<b>s-1</b>
<b>SAFETY PRECAUTIONS.....</b>	<b>s-1</b>
<b>1 KAREL LANGUAGE OVERVIEW .....</b>	<b>1</b>
1.1 KAREL PROGRAMMING LANGUAGE .....	1
1.1.1 Creating a Program .....	2
1.1.2 Translating a Program .....	2
1.1.3 Loading Program Logic and Data.....	3
1.1.4 Executing a Program .....	4
1.1.5 Execution History.....	4
1.1.6 Program Structure.....	4
1.2 SYSTEM SOFTWARE.....	5
1.2.1 Software Components.....	5
1.2.2 Supported Robots .....	6
1.3 CONTROLLER .....	6
1.3.1 Memory.....	6
1.3.2 Input/Output System .....	8
1.3.3 User Interface Devices.....	8
1.3.4 Frames.....	9
<b>2 LANGUAGE ELEMENTS .....</b>	<b>11</b>
2.1 LANGUAGE COMPONENTS .....	11
2.1.1 Character Set .....	11
2.1.2 Operators .....	13
2.1.3 Reserved Words .....	13
2.1.4 User-Defined Identifiers .....	14
2.1.5 Labels .....	15
2.1.6 Predefined Identifiers .....	15
2.1.7 System Variables .....	17
2.1.8 Comments .....	17
2.2 TRANSLATOR DIRECTIVES.....	18
2.3 DATA TYPES .....	19
2.4 USER-DEFINED DATA TYPES AND STRUCTURES .....	20
2.4.1 User-Defined Data Types .....	20
2.4.2 User-Defined Data Structures .....	22
2.5 ARRAYS .....	23
2.5.1 Multi-Dimensional Arrays .....	24
2.5.2 Variable-Sized Arrays .....	25
<b>3 USE OF OPERATORS .....</b>	<b>27</b>
3.1 EXPRESSIONS AND ASSIGNMENTS .....	27
3.1.1 Rule for Expressions and Assignments .....	27
3.1.2 Evaluation of Expressions and Assignments .....	27
3.1.3 Variables and Expressions .....	29
3.2 OPERATIONS .....	29
3.2.1 Arithmetic Operations .....	30
3.2.2 Relational Operations .....	31
3.2.3 Boolean Operations .....	32
3.2.4 Special Operations .....	32

<b>4 PROGRAM CONTROL .....</b>	<b>37</b>
4.1 PROGRAM CONTROL STRUCTURES .....	37
4.1.1 Alteration Control Structures .....	37
4.1.2 Looping Control Statements .....	37
4.1.3 Unconditional Branch Statement .....	38
4.1.4 Execution Control Statements .....	38
4.1.5 Condition Handlers .....	38
<b>5 ROUTINES .....</b>	<b>39</b>
5.1 ROUTINE EXECUTION .....	39
5.1.1 Declaring Routines .....	39
5.1.2 Invoking Routines .....	41
5.1.3 Returning from Routines .....	42
5.1.4 Scope of Variables .....	44
5.1.5 Parameters and Arguments .....	45
5.1.6 Stack Usage .....	48
5.2 BUILT- IN ROUTINES .....	50
<b>6 CONDITION HANDLERS .....</b>	<b>55</b>
6.1 CONDITION HANDLER OPERATIONS .....	56
6.1.1 Global Condition Handlers .....	56
6.2 CONDITIONS .....	58
6.2.1 Port_Id Conditions .....	59
6.2.2 Relational Conditions .....	59
6.2.3 System and Program Event Conditions .....	60
6.3 ACTIONS .....	62
6.3.1 Assignment Actions .....	62
6.3.2 Motion Related Actions .....	63
6.3.3 Routine Call Actions .....	64
6.3.4 Miscellaneous Actions .....	64
<b>7 FILE INPUT/OUTPUT OPERATIONS .....</b>	<b>67</b>
7.1 FILE VARIABLES .....	67
7.2 OPEN FILE STATEMENT .....	68
7.2.1 Setting File and Port Attributes .....	68
7.2.2 File String .....	73
7.2.3 Usage String .....	74
7.3 CLOSE FILE STATEMENT .....	76
7.4 READ STATEMENT .....	76
7.5 WRITE STATEMENT .....	78
7.6 INPUT/OUTPUT BUFFER .....	78
7.7 FORMATTING TEXT (ASCII) INPUT/OUTPUT .....	79
7.7.1 Formatting INTEGER Data Items .....	80
7.7.2 Formatting REAL Data Items .....	82
7.7.3 Formatting BOOLEAN Data Items .....	83
7.7.4 Formatting STRING Data Items .....	85
7.7.5 Formatting VECTOR Data Items .....	87
7.7.6 Formatting Positional Data Items .....	87
7.8 FORMATTING BINARY INPUT/OUTPUT .....	88
7.8.1 Formatting INTEGER Data Items .....	89
7.8.2 Formatting REAL Data Items .....	90
7.8.3 Formatting BOOLEAN Data Items .....	90

7.8.4 Formatting STRING Data Items .....	90
7.8.5 Formatting VECTOR Data Items .....	91
7.8.6 Formatting POSITION Data Items .....	91
7.8.7 Formatting XYZWPR Data Items .....	91
7.8.8 Formatting XYZWPREXT Data Items .....	92
7.8.9 Formatting JOINTPOS Data Items .....	92
<b>7.9 USER INTERFACE TIPS .....</b>	<b>92</b>
7.9.1 USER Menu on the Teach Pendant .....	92
7.9.2 USER Menu on the CRT/KB .....	93
<b>8 POSITION DATA .....</b>	<b>95</b>
8.1 POSITIONAL DATA .....	95
8.2 FRAMES OF REFERENCE .....	96
8.2.1 World Frame .....	97
8.2.2 User Frame (UFRAME) .....	97
8.2.3 Tool Definition (UTOOL) .....	97
8.2.4 Using Frames in the Teach Pendant Editor (TP) .....	98
8.3 JOG COORDINATE SYSTEMS .....	98
<b>9 TEACHING KAREL VARIABLES .....</b>	<b>101</b>
9.1 KAREL Positions .....	101
9.1.1 Teaching KAREL Positions .....	101
9.2 KAREL Paths .....	104
9.2.1 Teaching KAREL Paths .....	105
9.3 KAREL Variables .....	111
9.3.1 Modifying KAREL Variables .....	111
<b>10 FILE SYSTEM .....</b>	<b>113</b>
10.1 FILE SPECIFICATION .....	113
10.1.1 Device Name .....	114
10.1.2 File Name .....	115
10.1.3 File Type .....	116
10.2 STORAGE DEVICE ACCESS .....	117
10.2.1 Storage Device Access Overview .....	117
10.2.2 Memory File Devices .....	122
10.2.3 Virtual Devices .....	123
10.2.4 File Pipes .....	124
10.3 FILE ACCESS .....	128
10.4 FORMATTING XML INPUT .....	128
10.4.1 Installation Sequence .....	129
10.4.2 Example KAREL Program Referencing an XML File .....	129
10.4.3 Parse Errors .....	133
10.5 MEMORY DEVICE .....	133
<b>11 DICTIONARIES AND FORMS .....</b>	<b>137</b>
11.1 CREATING USER DICTIONARIES .....	137
11.1.1 Dictionary Syntax .....	137
11.1.2 Dictionary Element Number .....	138
11.1.3 Dictionary Element Name .....	138
11.1.4 Dictionary Cursor Positioning .....	139
11.1.5 Dictionary Element Text .....	139
11.1.6 Dictionary Reserved Word Commands .....	141
11.1.7 Character Codes .....	143

11.1.8 Nesting Dictionary Elements .....	143
11.1.9 Dictionary Comment .....	143
11.1.10 Generating a KAREL Constant File .....	143
11.1.11 Compressing and Loading Dictionaries on the Controller .....	144
11.1.12 Accessing Dictionary Elements from a KAREL Program .....	145
<b>11.2 CREATING USER FORMS .....</b>	<b>145</b>
11.2.1 Form Syntax .....	146
11.2.2 Form Attributes .....	148
11.2.3 Form Title and Menu Label .....	148
11.2.4 Form Menu Text .....	149
11.2.5 Form Selectable Menu Item .....	150
11.2.6 Edit Data Item .....	151
11.2.7 Dynamic Forms using Tree View.....	157
11.2.8 Non-Selectable Text .....	157
11.2.9 Display Only Data Items .....	157
11.2.10 Cursor Position Attributes .....	157
11.2.11 Form Reserved Words and Character Codes .....	158
11.2.12 Form Function Key Element Name or Number .....	159
11.2.13 Form Function Key Using a Variable .....	160
11.2.14 Form Help Element Name or Number .....	161
11.2.15 Teach Pendant Form Screen .....	161
11.2.16 CRT/KB Form Screen .....	162
11.2.17 Form File Naming Convention .....	162
11.2.18 Compressing and Loading Forms on the Controller .....	163
11.2.19 Displaying a Form .....	164
<b>12 SOCKET MESSAGING .....</b>	<b>173</b>
<b>12.1 SYSTEM REQUIREMENTS .....</b>	<b>173</b>
12.1.1 Software Requirements .....	173
12.1.2 Hardware Requirements .....	173
<b>12.2 CONFIGURING THE SOCKET MESSAGING OPTION .....</b>	<b>173</b>
12.2.1 Setting up a Server Tag .....	174
12.2.1.1 Setting up a Server Tag .....	174
12.2.2 Setting up a Client Tag .....	176
12.2.2.1 Setting up a ClientTag .....	176
<b>12.3 SOCKET MESSAGING AND KAREL .....</b>	<b>178</b>
12.3.1 MSG_CONNECT(string, integer) .....	179
12.3.2 MSG_DISCO(string, integer) .....	179
12.3.3 MSG_PING(string, integer) .....	179
12.3.4 Exchanging Data during a Socket Messaging Connection .....	180
<b>12.4 NETWORK PERFORMANCE .....</b>	<b>180</b>
12.4.1 Guidelines for a Good Implementation .....	180
<b>12.5 PROGRAMMING EXAMPLES .....</b>	<b>180</b>
12.5.1 A KAREL Client Application .....	181
12.5.2 A KAREL Server Application .....	183
12.5.3 ANSI C Loopback Client Example .....	184
<b>13 DATA TRANSFER BETWEEN ROBOTS OVER ETHERNET (DTBR)..</b>	<b>187</b>
<b>13.1 TERMINOLOGY.....</b>	<b>187</b>
<b>13.2 SETUP.....</b>	<b>187</b>
13.2.1 TCP/IP Setup.....	188
<b>13.3 TCP/IP SETUP FOR ROBOGUIDE.....</b>	<b>189</b>
<b>13.4 STANDARD DATA TRANSFER PROGRAMS.....</b>	<b>190</b>

13.4.1 RGETNREG: Program to Get Numeric Register.....	191
13.4.2 RSETNREG: Program to Set Numeric Register.....	192
13.4.3 RGETPREG: Program to Get Position Register.....	192
13.4.4 RSETPREG: Program to Set Position Register.....	194
<b>13.5 ERROR RECOVERY.....</b>	<b>195</b>
13.6 KAREL BUILT-INS.....	196
13.7 TIME OUT AND RETRY.....	197
13.8 LIMITATIONS.....	197
13.9 TROUBLESHOOTING.....	198
<b>14 SYSTEM VARIABLES .....</b>	<b>201</b>
14.1 ACCESS RIGHTS .....	201
14.2 STORAGE .....	201
<b>15 KAREL COMMAND LANGUAGE (KCL) .....</b>	<b>203</b>
15.1 COMMAND FORMAT .....	203
15.1.1 Default Program .....	203
15.1.2 Variables and Data Types .....	203
15.2 PROGRAM CONTROL COMMANDS .....	204
15.3 ENTERING COMMANDS .....	204
15.3.1 Abbreviations .....	205
15.3.2 Error Messages .....	205
15.3.3 Subdirectories .....	205
15.4 COMMAND PROCEDURES .....	205
15.4.1 Command Procedure Format .....	205
15.4.2 Creating Command Procedures .....	206
15.4.3 Error Processing .....	206
15.4.4 Executing Command Procedures .....	206
<b>16 INPUT/OUTPUT SYSTEM .....</b>	<b>209</b>
16.1 USER-DEFINED SIGNALS .....	209
16.1.1 DIN and DOUT Signals .....	209
16.1.2 GIN and GOUT Signals .....	210
16.1.3 AIN and AOUT Signals .....	210
16.1.4 Hand Signals .....	211
16.2 SYSTEM-DEFINED SIGNALS .....	212
16.2.1 Robot Digital Input and Output Signals (RDI/RDO) .....	212
16.2.2 Operator Panel Input and Output Signals (OPIN/OPOUT) .....	212
16.2.3 Teach Pendant Input and Output Signals (TPIN/TPOUT) .....	219
16.3 SERIAL INPUT/OUTPUT .....	225
<b>17 MULTI-TASKING .....</b>	<b>229</b>
17.1 MULTI-TASKING TERMINOLOGY .....	229
17.2 INTERPRETER ASSIGNMENT .....	230
17.3 MOTION CONTROL .....	230
17.4 TASK SCHEDULING .....	231
17.4.1 Priority Scheduling .....	231
17.4.2 Time Slicing .....	232
17.5 STARTING TASKS .....	232
17.5.1 Running Programs from the User Operator Panel (UOP) PNS Signal .....	233
17.5.2 Child Tasks .....	233
17.6 TASK CONTROL AND MONITORING .....	233
17.6.1 From TPP Programs .....	233

17.6.2 From KAREL Programs .....	234
17.6.3 From KCL .....	234
<b>17.7 USING SEMAPHORES AND TASK SYNCHRONIZATION .....</b>	<b>234</b>
<b>17.8 USING QUEUES FOR TASK COMMUNICATIONS .....</b>	<b>239</b>

## APPENDIX

### A KAREL LANGUAGE ALPHABETICAL DESCRIPTION ..... 243

<b>A.1 - A - KAREL LANGUAGE DESCRIPTION .....</b>	<b>250</b>
A.1.1 ABORT Action .....	250
A.1.2 ABORT Condition .....	251
A.1.3 ABORT Statement .....	251
A.1.4 ABORT_TASK Built-In Procedure .....	252
A.1.5 ABS Built-In Function .....	252
A.1.6 ACOS Built-In Function .....	252
A.1.7 ACT_SCREEN Built-In Procedure .....	253
A.1.8 ACT_TBL Built-In Procedure.....	254
A.1.9 ADD_BYNAMEPC Built-In Procedure .....	255
A.1.10 ADD_DICT Built-In Procedure .....	256
A.1.11 ADD_INTPC Built-In Procedure .....	257
A.1.12 ADD_REALPC Built-In Procedure .....	258
A.1.13 ADD_STRINGPC Built-In Procedure .....	259
A.1.14 %ALPHABETIZE Translator Directive .....	260
A.1.15 APPEND_NODE Built-In Procedure .....	260
A.1.16 APPEND_QUEUE Built-In Procedure .....	261
A.1.17 APPROACH Built-In Function .....	261
A.1.18 ARRAY Data Type .....	262
A.1.19 ARRAY_LEN Built-In Function .....	263
A.1.20 ASIN Built-In Function .....	263
A.1.21 Assignment Action .....	264
A.1.22 Assignment Statement .....	265
A.1.23 ATAN2 Built-In Function .....	266
A.1.24 ATTACH Statement .....	267
A.1.25 ATT_WINDOW_D Built-In Procedure .....	267
A.1.26 ATT_WINDOW_S Built-In Procedure .....	268
A.1.27 AVL_POS_NUM Built-In Procedure .....	269
<b>A.2 - B - KAREL LANGUAGE DESCRIPTION .....</b>	<b>269</b>
A.2.1 BOOLEAN Data Type .....	269
A.2.2 BYNAME Built-In Function .....	270
A.2.3 BYTE Data Type .....	271
A.2.4 BYTES_AHEAD Built-In Procedure .....	271
A.2.5 BYTES_LEFT Built-In Function .....	272
<b>A.3 - C - KAREL LANGUAGE DESCRIPTION .....</b>	<b>274</b>
A.3.1 CALL_PROG Built-In Procedure .....	274
A.3.2 CALL_PROGLIN Built-In Procedure .....	274
A.3.3 CANCEL Action .....	275
A.3.4 CANCEL Statement .....	275
A.3.5 CANCEL_FILE Statement .....	276
A.3.6 CHECK_DICT Built-In Procedure .....	277
A.3.7 CHECK_EPOS Built-In Procedure .....	277
A.3.8 CHECK_NAME Built-In Procedure .....	278
A.3.9 CHR Built-In Function .....	278

A.3.10 CLEAR Built-In Procedure .....	279
A.3.11 CLEAR_SEMA Built-In Procedure .....	279
A.3.12 CLOSE FILE Statement .....	280
A.3.13 CLOSE HAND Statement .....	280
A.3.14 CLOSE_TPE Built-In Procedure .....	281
A.3.15 CLR_IO_STAT Built-In Procedure .....	281
A.3.16 CLR_PORT_SIM Built-In Procedure .....	282
A.3.17 CLR_POS_REG Built-In Procedure .....	282
A.3.18 %CMOSVARS Translator Directive .....	283
A.3.19 %CMOS2SHADOW Translator Directive .....	283
A.3.20 CNC_DYN_DISB Built-In Procedure .....	283
A.3.21 CNC_DYN_DISE Built-In Procedure .....	284
A.3.22 CNC_DYN_DISI Built-In Procedure .....	284
A.3.23 CNC_DYN_DISP Built-In Procedure .....	285
A.3.24 CNC_DYN_DISR Built-In Procedure .....	285
A.3.25 CNC_DYN_DISS Built-In Procedure .....	286
A.3.26 CNCL_STP_MTN Built-In Procedure .....	286
A.3.27 CNV_CNF_STRG Built-In Procedure .....	287
A.3.28 CNV_CONF_STR Built-In Procedure .....	288
A.3.29 CNV_INT_STR Built-In Procedure .....	288
A.3.30 CNV_JPOS_REL Built-In Procedure .....	289
A.3.31 CNV_REAL_STR Built-In Procedure .....	289
A.3.32 CNV_REL_JPOS Built-In Procedure .....	290
A.3.33 CNV_STR_CONF Built-In Procedure .....	291
A.3.34 CNV_STR_INT Built-In Procedure .....	291
A.3.35 CNV_STR_REAL Built-In Procedure .....	292
A.3.36 CNV_STR_TIME Built-In Procedure .....	292
A.3.37 CNV_TIME_STR Built-In Procedure .....	293
A.3.38 %COMMENT Translator Directive .....	293
A.3.39 COMPARE_FILE Built-in Procedure .....	294
A.3.40 CONDITION...ENDCONDITION Statement .....	296
A.3.41 CONFIG Data Type .....	297
A.3.42 CONNECT TIMER Statement .....	298
A.3.43 CONTINUE Action .....	298
A.3.44 CONTINUE Condition .....	299
A.3.45 CONT_TASK Built-In Procedure .....	299
A.3.46 COPY_FILE Built-In Procedure .....	300
A.3.47 COPY_PATH Built-In Procedure .....	301
A.3.48 COPY_QUEUE Built-In Procedure .....	302
A.3.49 COPY_TPE Built-In Procedure .....	303
A.3.50 COS Built-In Function .....	304
A.3.51 CR Input/Output Item .....	304
A.3.52 CREATE_TPE Built-In Procedure .....	305
A.3.53 CREATE_VAR Built-In Procedure .....	306
A.3.54 %CRTDEVICE Translator Directive .....	307
A.3.55 CURJPOS Built-In Function .....	308
A.3.56 CURPOS Built-In Function .....	308
A.3.57 CURR_PROG Built-In Function .....	309
<b>A.4 - D - KAREL LANGUAGE DESCRIPTION .....</b>	<b>309</b>
A.4.1 DAQ_CHECKP Built-In Procedure .....	310
A.4.2 DAQ_REGPIPE Built-In Procedure .....	310
A.4.3 DAQ_START Built-In Procedure .....	312
A.4.4 DAQ_STOP Built-In Procedure .....	313

A.4.5 DAQ_UNREG Built-In Procedure .....	314
A.4.6 DAQ_WRITE Built-In Procedure .....	315
A.4.7 %DEFGROUP Translator Directive .....	316
A.4.8 DEF_SCREEN Built-In Procedure .....	316
A.4.9 DEF_WINDOW Built-In Procedure .....	317
A.4.10 %DELAY Translator Directive .....	318
A.4.11 DELAY Statement .....	318
A.4.12 DELETE_FILE Built-In Procedure .....	319
A.4.13 DELETE_NODE Built-In Procedure .....	320
A.4.14 DELETE_QUEUE Built-In Procedure .....	320
A.4.15 DEL_INST_TPE Built-In Procedure .....	321
A.4.16 DET_WINDOW Built-In Procedure .....	321
A.4.17 DISABLE CONDITION Action .....	322
A.4.18 DISABLE CONDITION Statement .....	322
A.4.19 DISCONNECT TIMER Statement .....	323
A.4.20 DISCTRL_ALPH Built-In Procedure .....	324
A.4.21 DISCTRL_FORM Built-In Procedure .....	325
A.4.22 DISCTRL_LIST Built-In Procedure .....	327
A.4.23 DISCTRL_PLMN Built-In Procedure .....	328
A.4.24 DISCTRL_SBMN Built-In Procedure .....	329
A.4.25 DISCTRL_TBL Built-In Procedure .....	331
A.4.26 DISMOUNT_DEV Built-In Procedure .....	333
A.4.27 DISP_DAT_T Data Type .....	333
A.4.28 DOSFILE_INF Built-In Procedure.....	335
<b>A.5 - E - KAREL LANGUAGE DESCRIPTION .....</b>	<b>336</b>
A.5.1 ENABLE CONDITION Action .....	336
A.5.2 ENABLE CONDITION Statement .....	336
A.5.3 %ENVIRONMENT Translator Directive .....	337
A.5.4 ERR_DATA Built-In Procedure .....	338
A.5.5 ERROR Condition .....	339
A.5.6 EVAL Clause .....	340
A.5.7 EVENT Condition .....	340
A.5.8 EXP Built-In Function .....	341
<b>A.6 - F - KAREL LANGUAGE DESCRIPTION .....</b>	<b>341</b>
A.6.1 FILE Data Type .....	341
A.6.2 FILE_LIST Built-In Procedure .....	342
A.6.3 FOR...ENDFOR Statement .....	343
A.6.4 FORCE_LINK Built-In Procedure.....	344
A.6.5 FORCE_SPMENU Built-In Procedure .....	345
A.6.6 FORMAT_DEV Built-In Procedure .....	347
A.6.7 FRAME Built-In Function .....	348
A.6.8 FROM Clause .....	349
<b>A.7 - G - KAREL LANGUAGE DESCRIPTION .....</b>	<b>349</b>
A.7.1 GET_ATTR_PRG Built-In Procedure .....	349
A.7.2 GET_FILE_POS Built-In Function .....	351
A.7.3 GET_JPOS_REG Built-In Function .....	352
A.7.4 GET_JPOS_TPE Built-In Function .....	352
A.7.5 GET_PORT_ASG Built-in Procedure .....	353
A.7.6 GET_PORT_ATR Built-In Function .....	354
A.7.7 GET_PORT_CMT Built-In Procedure .....	356
A.7.8 GET_PORT_MOD Built-In Procedure .....	356
A.7.9 GET_PORT_SIM Built-In Procedure .....	358
A.7.10 GET_PORT_VAL Built-In Procedure .....	358

A.7.11 GET_POS_FRM Built-In Procedure .....	359
A.7.12 GET_POS_REG Built-In Function .....	359
A.7.13 GET_POS_TPE Built-In Function .....	360
A.7.14 GET_POS_TYP Built-In Procedure .....	361
A.7.15 GET_PREG_CMT Built-In-Procedure .....	361
A.7.16 GET_QUEUE Built-In Procedure .....	362
A.7.17 GET_REG Built-In Procedure .....	363
A.7.18 GET_REG_CMT Built-In Procedure.....	363
A.7.19 GET_SREG_CMT Built-In Procedure.....	364
A.7.20 GET_STR_REG Built-In Procedure.....	364
A.7.21 GET_TIME Built-In Procedure .....	365
A.7.22 GET_TPE_CMT Built-in Procedure .....	365
A.7.23 GET_TPE_PRM Built-in Procedure .....	366
A.7.24 GET_TSK_INFO Built-In Procedure .....	368
A.7.25 GET_USEC_SUB Built-In Procedure .....	369
A.7.26 GET_USEC_TIM Built-In Function .....	369
A.7.27 GET_VAR Built-In Procedure .....	370
A.7.28 GO TO Statement .....	373
<b>A.8 - H - KAREL LANGUAGE DESCRIPTION .....</b>	<b>374</b>
A.8.1 HOLD Action .....	374
A.8.2 HOLD Statement .....	374
<b>A.9 - I - KAREL LANGUAGE DESCRIPTION .....</b>	<b>375</b>
A.9.1 IF ... ENDIF Statement .....	375
A.9.2 IN Clause .....	376
A.9.3 %INCLUDE Translator Directive .....	376
A.9.4 INDEX Built-In Function .....	377
A.9.5INI_DYN_DISB Built-In Procedure .....	378
A.9.6INI_DYN_DISE Built-In Procedure .....	379
A.9.7INI_DYN_DISI Built-In Procedure .....	380
A.9.8INI_DYN_DISP Built-In Procedure .....	381
A.9.9INI_DYN_DISR Built-In Procedure .....	382
A.9.10INI_DYN_DISS Built-In Procedure .....	383
A.9.11INIT_QUEUE Built-In Procedure .....	384
A.9.12INIT_TBL Built-In Procedure .....	384
A.9.13IN_RANGE Built-In Function .....	393
A.9.14INSERT_NODE Built-In Procedure .....	393
A.9.15INSERT_QUEUE Built-In Procedure .....	394
A.9.16INTEGER Data Type .....	395
A.9.17INV Built-In Function .....	396
A.9.18IO_MOD_TYPE Built-In Procedure .....	396
A.9.19IO_STATUS Built-In Function .....	397
<b>A.10 - J - KAREL LANGUAGE DESCRIPTION .....</b>	<b>398</b>
A.10.1J_IN_RANGE Built-In Function .....	398
A.10.2JOINTPOS Data Type .....	399
A.10.3JOINT2POS Built-In Function .....	399
<b>A.11 - K - KAREL LANGUAGE DESCRIPTION .....</b>	<b>400</b>
A.11.1KCL Built-In Procedure .....	400
A.11.2KCL_NO_WAIT Built-In Procedure .....	401
A.11.3KCL_STATUS Built-In Procedure .....	402
<b>A.12 - L - KAREL LANGUAGE DESCRIPTION .....</b>	<b>402</b>
A.12.1LN Built-In Function .....	402
A.12.2LOAD Built-In Procedure .....	403
A.12.3LOAD_STATUS Built-In Procedure .....	404

A.12.4 LOCK_GROUP Built-In Procedure .....	404
A.12.5 %LOCKGROUP Translator Directive .....	405
<b>A.13 - M - KAREL LANGUAGE DESCRIPTION .....</b>	<b>406</b>
A.13.1 MIRROR Built-In Function .....	406
A.13.2 MODIFY_QUEUE Built-In Procedure .....	407
A.13.3 MOTION_CTL Built-In Function .....	408
A.13.4 MOUNT_DEV Built-In Procedure .....	409
A.13.5 MOVE_FILE Built-In Procedure .....	409
A.13.6 MSG_CONNECT Built-In Procedure .....	410
A.13.7 MSG_DISCO Built-In Procedure .....	411
A.13.8 MSG_PING Built-In Procedure .....	412
<b>A.14 - N - KAREL LANGUAGE DESCRIPTION .....</b>	<b>412</b>
A.14.1 NOABORT Action .....	412
A.14.2 %NOABORT Translator Directive .....	413
A.14.3 %NOBUSYLAMP Translator Directive .....	413
A.14.4 NODE_SIZE Built-In Function .....	413
A.14.5 %NOLOCKGROUP Translator Directive .....	415
A.14.6 NOMESSAGE Action .....	415
A.14.7 NOPAUSE Action .....	416
A.14.8 %NOPAUSE Translator Directive .....	416
A.14.9 %NOPAUSESHFT Translator Directive .....	417
<b>A.15 - O - KAREL LANGUAGE DESCRIPTION .....</b>	<b>417</b>
A.15.1 OPEN FILE Statement .....	417
A.15.2 OPEN HAND Statement .....	418
A.15.3 OPEN_TPE Built-In Procedure .....	418
A.15.4 ORD Built-In Function .....	419
A.15.5 ORIENT Built-In Function .....	420
<b>A.16 - P - KAREL LANGUAGE DESCRIPTION .....</b>	<b>420</b>
A.16.1 PATH Data Type .....	420
A.16.2 PATH_LEN Built-In Function .....	422
A.16.3 PAUSE Action .....	422
A.16.4 PAUSE Condition .....	423
A.16.5 PAUSE Statement .....	423
A.16.6 PAUSE_TASK Built-In Procedure .....	424
A.16.7 PEND_SEMA Built-In Procedure .....	425
A.16.8 PIPE_CONFIG Built-In Procedure .....	426
A.16.9 POP_KEY_RD Built-In Procedure .....	426
A.16.10 Port_Id Action .....	427
A.16.11 Port_Id Condition .....	427
A.16.12 POS Built-In Function .....	428
A.16.13 POS2JOINT Built-In Function .....	429
A.16.14 POS_REG_TYPE Built-In Procedure .....	430
A.16.15 POSITION Data Type .....	431
A.16.16 POST_ERR Built-In Procedure .....	432
A.16.17 POST_ERR_L Built-In Procedure .....	432
A.16.18 POST_SEMA Built-In Procedure .....	433
A.16.19 PRINT_FILE Built-In Procedure .....	434
A.16.20 %PRIORITY Translator Directive .....	434
A.16.21 PROG_BACKUP Built-In Procedure .....	435
A.16.22 PROG_CLEAR Built-In Procedure .....	436
A.16.23 PROG_LIST Built-In Procedure .....	438
A.16.24 PROG_RESTORE Built-In Procedure .....	439
A.16.25 PROGRAM Statement .....	440

A.16.26 PULSE Action .....	441
A.16.27 PULSE Statement .....	441
A.16.28 PURGE CONDITION Statement .....	442
A.16.29 PURGE_DEV Built-In Procedure .....	443
A.16.30 PUSH_KEY_RD Built-In Procedure .....	443
<b>A.17 - Q - KAREL LANGUAGE DESCRIPTION .....</b>	<b>444</b>
A.17.1 QUEUE_ATTACH Built-in Procedure.....	444
A.17.2 QUEUE_TYPE Data Type .....	446
<b>A.18 - R - KAREL LANGUAGE DESCRIPTION .....</b>	<b>446</b>
A.18.1 READ Statement .....	446
A.18.2 READ_DICT Built-In Procedure .....	447
A.18.3 READ_DICT_V Built-In-Procedure .....	448
A.18.4 READ_KB Built-In Procedure .....	449
A.18.5 REAL Data Type .....	452
A.18.6 Relational Condition .....	453
A.18.7 RELAX HAND Statement .....	454
A.18.8 RELEASE Statement .....	455
A.18.9 REMOVE_DICT Built-In Procedure .....	455
A.18.10 RENAME_FILE Built-In Procedure .....	456
A.18.11 RENAME_VAR Built-In Procedure .....	456
A.18.12 RENAME_VARS Built-In Procedure .....	457
A.18.13 REPEAT ... UNTIL Statement .....	457
A.18.14 RESET Built-In Procedure .....	458
A.18.15 RESUME Action .....	458
A.18.16 RESUME Statement .....	459
A.18.17 RETURN Statement .....	460
A.18.18 RGET_PORTCMT Built-In Routine.....	460
A.18.19 RGET_PORTSIM Built-In Routine.....	461
A.18.20 RGET_PORTVAL Built-In Routine.....	462
A.18.21 RGET_PREGCMT Built-In Routine.....	462
A.18.22 RGET_REG Built-In Routine.....	463
A.18.23 RGET_REG_CMT Built-In Routine.....	464
A.18.24 RGET_SREGCMT Built-in Routine.....	465
A.18.25 RGET_STR_REG Built-In Routine.....	465
A.18.26 RMCN_ALERT Built-In Routine.....	466
A.18.27 RMCN_SEND Built-in Routine.....	467
A.18.28 RNUMREG_RECV Built-In Routine.....	468
A.18.29 RNUMREG_SEND Built-In Routine.....	469
A.18.30 ROUND Built-In Function .....	470
A.18.31 ROUTINE Statement .....	470
A.18.32 RPREG_RECV Built-In Routine.....	471
A.18.33 RPREG_SEND Built-in Routine.....	472
A.18.34 RSET_INT_REG Built-in Routine.....	473
A.18.35 RSET_PORTCMT Built-in Routine.....	474
A.18.36 RSET_PORTSIM Built-in Routine.....	475
A.18.37 RSET_PORTVAL Built-in Routine.....	475
A.18.38 RSET_PREGCMT Built-in Routine.....	476
A.18.39 RSET_REALREG Built-in Routine.....	477
A.18.40 RSET_REG_CMT Built-In Routine.....	477
A.18.41 RSET_SREGCMT Built-in Routine.....	478
A.18.42 RSET_STR_REG Built-in Routine.....	479
A.18.43 RUN_TASK Built-In Procedure .....	479
<b>A.19 - S - KAREL LANGUAGE DESCRIPTION .....</b>	<b>480</b>

A.19.1 SAVE Built-In Procedure .....	481
A.19.2 SAVE_DRAM Built-In Procedure .....	481
A.19.3 SELECT ... ENDSELECT Statement .....	482
A.19.4 SELECT_TPE Built-In Procedure .....	482
A.19.5 SEMA_COUNT Built-In Function .....	483
A.19.6 SEMAPHORE Condition .....	483
A.19.7 SEND_DATAPC Built-In Procedure .....	484
A.19.8 SEND_EVENTPC Built-In Procedure .....	485
A.19.9 SET_ATTR_PRG Built-In Procedure .....	485
A.19.10 SET_CURSOR Built-In Procedure .....	486
A.19.11 SET_EPOS_REG Built-In Procedure .....	487
A.19.12 SET_EPOS_TPE Built-In Procedure .....	488
A.19.13 SET_FILE_ATR Built-In Procedure .....	489
A.19.14 SET_FILE_POS Built-In Procedure .....	489
A.19.15 SET_INT_REG Built-In Procedure .....	490
A.19.16 SET_JPOS_REG Built-In Procedure .....	491
A.19.17 SET_JPOS_TPE Built-In Procedure .....	491
A.19.18 SET_LANG Built-In Procedure .....	492
A.19.19 SET_PERCH Built-In Procedure .....	493
A.19.20 SET_PORT_ASG Built-In Procedure .....	493
A.19.21 SET_PORT_ATR Built-In Function .....	494
A.19.22 SET_PORT_CMT Built-In Procedure .....	496
A.19.23 SET_PORT_MOD Built-In Procedure .....	497
A.19.24 SET_PORT_SIM Built-In Procedure .....	497
A.19.25 SET_PORT_VAL Built-In Procedure .....	498
A.19.26 SET_POS_REG Built-In Procedure .....	499
A.19.27 SET_POS_TPE Built-In Procedure .....	500
A.19.28 SET_PREG_CMT Built-In-Procedure .....	500
A.19.29 SET_REAL_REG Built-In Procedure .....	501
A.19.30 SET_REG_CMT Built-In-Procedure .....	501
A.19.31 SET_SREG_CMT Built-in Procedure .....	501
A.19.32 SET_STR_REG Built-in Procedure .....	502
A.19.33 SET_TIME Built-In Procedure .....	502
A.19.34 SET_TPE_CMT Built-In Procedure .....	503
A.19.35 SET_TRNS_TPE Built-In Procedure .....	504
A.19.36 SET_TSK_ATTR Built-In Procedure .....	504
A.19.37 SET_TSK_NAME Built-In Procedure .....	505
A.19.38 SET_VAR Built-In Procedure .....	506
A.19.39 %SHADOWVARS Translator Directive .....	508
A.19.40 SHORT Data Type .....	508
A.19.41 SIGNAL EVENT Action .....	509
A.19.42 SIGNAL EVENT Statement .....	509
A.19.43 SIGNAL SEMAPHORE Action .....	509
A.19.44 SIN Built-In Function .....	510
A.19.45 SQRT Built-In Function .....	510
A.19.46 %STACKSIZE Translator Directive .....	511
A.19.47 STD_PTH_NODE Data Type .....	511
A.19.48 STOP Action .....	511
A.19.49 STOP Statement .....	512
A.19.50 STRING Data Type .....	512
A.19.51 STR_LEN Built-In Function .....	513
A.19.52 STRUCTURE Data Type .....	514
A.19.53 SUB_STR Built-In Function .....	514

A.20 - T - KAREL LANGUAGE DESCRIPTION .....	515
A.20.1 TAN Built-In Function .....	515
A.20.2 %TIMESLICE Translator Directive .....	515
A.20.3 %TPMOTION Translator Directive .....	516
A.20.4 TRANSLATE Built-In Procedure .....	516
A.20.5 TRUNC Built-In Function .....	517
A.21 - U - KAREL LANGUAGE DESCRIPTION .....	518
A.21.1 UNHOLD Action .....	518
A.21.2 UNHOLD Statement .....	518
A.21.3 UNINIT Built-In Function .....	519
A.21.4 %UNINITVARS Translator Directive .....	519
A.21.5 UNLOCK_GROUP Built-In Procedure .....	519
A.21.6 UNPAUSE Action .....	521
A.21.7 UNPOS Built-In Procedure .....	521
A.21.8 USING ... ENDUSING Statement .....	522
A.22 - V - KAREL LANGUAGE DESCRIPTION .....	522
A.22.1 V_ACQ_VAMAP iRVision Built-In Procedure.....	522
A.22.2 V_ADJ_2D iRVision Built-In Procedure.....	523
A.22.3 V_CAM_CALIB iRVision Built-In Procedure .....	524
A.22.4 V_CAM_CHECK iRVision Built-In Procedure.....	525
A.22.5 V_CLR_VAMAP iRVision Built-In Procedure.....	526
A.22.6 V_CSAPI_GETVALUE iRVision Built-In Procedure.....	526
A.22.7 V_CSAPI_NUMSET iRVision Built-In Procedure.....	527
A.22.8 V_CSAPI_RESETDATA iRVision Built-In Procedure.....	527
A.22.9 V_CSAPI_SAVEDATA iRVision Built-In Procedure.....	528
A.22.10 V_CSAPI_SETVALUE iRVision Built-In Procedure.....	528
A.22.11 V_CSAPI_TESTRUN iRVision Built-In Procedure.....	529
A.22.12 V_DISPLAY4D iRVision Built-In Procedure.....	530
A.22.13 V_FIND_VIEW iRVision Built-In Procedure.....	530
A.22.14 V_FIND_VLINE iRVision Built-In Procedure.....	531
A.22.15 V_GET_FOUND iRVision Built-In Procedure.....	534
A.22.16 V_GET_OFFSET iRVision Built-In Procedure .....	535
A.22.17 V_GET_PASSFL iRVision Built-In Procedure .....	536
A.22.18 V_GET_READ iRVision Built-In Procedure .....	537
A.22.19 V_GET_VPARAM iRVision Built-In Procedure .....	538
A.22.20 V_IRCONNECT iRVision Built-In Procedure .....	539
A.22.21 V_LED_OFF iRVision Built-In Procedure .....	540
A.22.22 V_LED_ON iRVision Built-In Procedure .....	540
A.22.23 V_OVERRIDE iRVision Built-In Procedure .....	541
A.22.24 V_RUN_FIND iRVision Built-In Procedure .....	541
A.22.25 V_SAVE_IMREG iRVision Built-In Procedure .....	543
A.22.26 V_SET_REF iRVision Built-In Procedure .....	544
A.22.27 V_SNAP_VIEW iRVision Built-In Procedure .....	545
A.22.28 VAR_INFO Built-In Procedure .....	546
A.22.29 VAR_LIST Built-In Procedure .....	548
A.22.30 VECTOR Data Type .....	550
A.22.31 VOL_SPACE Built-In Procedure .....	550
A.22.32 VREG_FND_POS iRVision Built-in Procedure .....	552
A.22.33 VREG_OFFSET iRVision Built-in Procedure .....	552
A.22.34 VT_ACK_QUEUE iRVision Built-In Procedure .....	553
A.22.35 VT_CLR_QUEUE iRVision Built-In Procedure .....	553
A.22.36 VT_DELETE_PQ iRVision Built-In Procedure .....	554
A.22.37 VT_GET_AREID iRVision Built-In Procedure .....	556

A.22.38 VT_GET_FOUND iRVision Built-In Procedure.....	556
A.22.39 VT_GET_LINID iRVision Built-In Procedure.....	557
A.22.40 VT_GET_PFRT iRVision Built-In Procedure.....	557
A.22.41 VT_GET_QUEUE iRVision Built-In Procedure.....	559
A.22.42 VT_GET_TIME iRVision Built-In Procedure.....	560
A.22.43 VT_GET_TRYID iRVision Built-In Procedure.....	560
A.22.44 VT_PUT_QUE2 iRVision Built-In Procedure.....	561
A.22.45 VT_PUT_QUEUE iRVision Built-In Procedure.....	562
A.22.46 VT_READ_PQ iRVision Built-In Procedure.....	562
A.22.47 VT_SET_FLAG iRVision Built-In Procedure.....	567
A.22.48 VT_SET_LDBAL iRVision Built-In Procedure.....	568
A.22.49 VT_WRITE_PQ iRVision Built-In Procedure.....	569
<b>A.23 - W - KAREL LANGUAGE DESCRIPTION .....</b>	<b>571</b>
A.23.1 WAIT FOR Statement .....	571
A.23.2 WHEN Clause .....	571
A.23.3 WHILE...ENDWHILE Statement .....	572
A.23.4 WITH Clause .....	572
A.23.5 WRITE Statement .....	573
A.23.6 WRITE_DICT Built-In Procedure .....	573
A.23.7 WRITE_DICT_V Built-In Procedure .....	574
<b>A.24 - X - KAREL LANGUAGE DESCRIPTION .....</b>	<b>575</b>
A.24.1 XML_ADDTAG Built-In Procedure .....	575
A.24.2 XML_GETDATA Built-In Procedure .....	576
A.24.3 XML_REMTAG Built-In Procedure .....	577
A.24.4 XML_SCAN Built-In Procedure .....	577
A.24.5 XML_SETVAR Built-In Procedure .....	578
A.24.6 XYZWPR Data Type .....	579
A.24.7 XYZWPREXT Data Type .....	579
<b>A.25 - Y - KAREL LANGUAGE DESCRIPTION .....</b>	<b>580</b>
<b>A.26 - Z - KAREL LANGUAGE DESCRIPTION .....</b>	<b>580</b>
<b>B KAREL EXAMPLE PROGRAMS .....</b>	<b>581</b>
B.1 SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING .....	585
B.2 COPYING PATH VARIABLES .....	594
B.3 SAVING DATA TO THE DEFAULT DEVICE .....	602
B.4 STANDARD ROUTINES .....	604
B.5 USING REGISTER BUILT-INS .....	606
B.6 POSITION DATA SET AND CONDITION HANDLERS PROGRAM.....	610
B.7 LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS .....	614
B.8 USING THE FILE AND DEVICE BUILT-INS .....	619
B.9 USING DYNAMIC DISPLAY BUILT-INS .....	622
B.10 MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES .....	630
B.11 DISPLAYING A LIST FROM A DICTIONARY FILE .....	632
B.11.1 Dictionary Files .....	640
B.12 USING THE DISCTRL_ALPHA BUILT-IN .....	641
B.12.1 Dictionary Files .....	644
B.13 APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM .....	644
B.14 iRVISION CUSTOM SCREEN USING V_CSAPI BUILT-INS.....	651
<b>C KCL COMMAND ALPHABETICAL DESCRIPTION .....</b>	<b>655</b>
C.1 ABORT command .....	655
C.2 APPEND FILE command .....	656

C.3 APPEND NODE command .....	656
C.4 CHDIR command .....	657
C.5 CLEAR ALL command .....	657
C.6 CLEAR BREAK CONDITION command .....	658
C.7 CLEAR BREAK PROGRAM command .....	658
C.8 CLEAR DICT command .....	658
C.9 CLEAR PROGRAM command .....	659
C.10 CLEAR VARS command .....	659
C.11 COMPRESS DICT command .....	660
C.12 COMPRESS FORM command .....	660
C.13 CONTINUE command .....	660
C.14 COPY FILE command .....	661
C.15 CREATE VARIABLE command .....	661
C.16 DELETE FILE command .....	662
C.17 DELETE NODE command .....	663
C.18 DELETE VARIABLE command .....	663
C.19 DIRECTORY command .....	663
C.20 DISABLE BREAK PROGRAM command .....	664
C.21 DISABLE CONDITION command .....	664
C.22 DISMOUNT command .....	665
C.23 EDIT command .....	665
C.24 ENABLE BREAK PROGRAM .....	665
C.25 ENABLE CONDITION command .....	666
C.26 FORMAT command .....	666
C.27 HELP command .....	666
C.28 HOLD command .....	667
C.29 INSERT NODE command .....	667
C.30 LOAD ALL command .....	668
C.31 LOAD DICT command .....	668
C.32 LOAD FORM command .....	669
C.33 LOAD MASTER command .....	669
C.34 LOAD PROGRAM command .....	669
C.35 LOAD SERVO command .....	670
C.36 LOAD SYSTEM command .....	670
C.37 LOAD TP command .....	671
C.38 LOAD VARS command .....	672
C.39 LOGOUT command .....	673
C.40 MKDIR command .....	673
C.41 MOUNT command .....	673
C.42 MOVE FILE command .....	674
C.43 PAUSE command .....	674
C.44 PURGE command .....	675
C.45 PRINT command .....	675
C.46 RECORD command .....	676
C.47 RENAME FILE command .....	676
C.48 RENAME VARIABLE command .....	676
C.49 RENAME VARS command .....	677
C.50 RESET command .....	678
C.51 RMDIR command .....	678
C.52 RUN command .....	678
C.53 RUNCF command .....	679
C.54 SAVE MASTER command .....	679
C.55 SAVE SERVO command .....	680

C.56 SAVE SYSTEM command .....	680
C.57 SAVE TP command .....	680
C.58 SAVE VARS command .....	681
C.59 SET BREAK CONDITION command .....	682
C.60 SET BREAK PROGRAM command .....	682
C.61 SET CLOCK command .....	683
C.62 SET DEFAULT command .....	683
C.63 SET GROUP command .....	684
C.64 SET LANGUAGE command .....	684
C.65 SET LOCAL VARIABLE command .....	684
C.66 SET PORT command .....	685
C.67 SET TASK command .....	685
C.68 SET TRACE command .....	686
C.69 SET VARIABLE command .....	686
C.70 SET VERIFY command .....	687
C.71 SHOW BREAK command .....	687
C.72 SHOW BUILTINS command .....	688
C.73 SHOW CONDITION command .....	688
C.74 SHOW CLOCK command .....	688
C.75 SHOW CURPOS command .....	689
C.76 SHOW DEFAULT command .....	689
C.77 SHOW DEVICE command .....	689
C.78 SHOW DICTS command .....	689
C.79 SHOW GROUP command .....	690
C.80 SHOW HISTORY command .....	690
C.81 SHOW LANG command .....	690
C.82 SHOW LANGS command .....	690
C.83 SHOW LOCAL VARIABLE command .....	690
C.84 SHOW LOCAL VARS command .....	691
C.85 SHOW MEMORY command .....	692
C.86 SHOW PROGRAM command .....	692
C.87 SHOW PROGRAMS command .....	692
C.88 SHOW SYSTEM command .....	693
C.89 SHOW TASK command .....	693
C.90 SHOW TASKS command .....	693
C.91 SHOW TRACE command .....	694
C.92 SHOW TYPES command .....	694
C.93 SHOW VARIABLE command .....	694
C.94 SHOW VARS command .....	695
C.95 SHOW data_type command .....	695
C.96 SIMULATE command .....	696
C.97 SKIP command .....	696
C.98 STEP OFF command .....	697
C.99 STEP ON command .....	697
C.100 TRANSLATE command .....	698
C.101 TYPE command .....	698
C.102 UNSIMULATE command .....	698
C.103 WAIT command .....	699

<b>D CHARACTER CODES.....</b>	<b>701</b>
-------------------------------	------------

<b>E SYNTAX DIAGRAMS .....</b>	<b>709</b>
--------------------------------	------------

<b>F TRANSLATING BY ROBOGUIDE.....</b>	<b>735</b>
F.1 TRANSLATE KAREL PROGRAMS.....	735
F.1.1 Sample of Newly Adding a KAREL Program.....	735
F.1.2 Sample of Adding an Existing KAREL Program.....	736
F.2 TRANSLATE A DICTIONARY.....	738
F.2.1 Sample of Translating Newly Created Dictionaries.....	738
F.2.2 Sample of Translating Added Existing Dictionaries.....	739
F.2.3 Dictionary Name and Languages.....	741
F.3 LOAD ON THE ROBOT CONTROLLER.....	741

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# ORIGINAL INSTRUCTIONS

Thank you very much for purchasing a FANUC robot.

Before using the robot, be sure to read the, *FANUC Robot series SAFETY HANDBOOK (B-80687EN)* and understand its contents.

- No part of this manual may be reproduced, copied, downloaded, translated into another language, published in any physical or electronic format, or in any other form, including the internet, or transmitted in whole or in part in any way without the prior written consent of FANUC or FANUC America Corporation.
- The appearance and specifications of this product are subject to change without notice.

The products in this manual are controlled based on Japan's "Foreign Exchange and Foreign Trade Law". The export from Japan may be subject to an export license by the government of Japan. Further, re-export to another country may be subject to the license of the government of the country from where the product is re-exported. Furthermore, the product may also be controlled by re-export regulations of the United States government. Should you wish to export or re-export these products, please contact FANUC for advice.

In this manual, we endeavor to include all pertinent matters. There are, however, a very large number of operations that must not or cannot be performed. Please assume that any operations that are not explicitly described as being possible are not possible.

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# SAFETY PRECAUTIONS

---

This chapter describes the precautions which must be followed to enable the safe use of the robot. Before using the robot, be sure to read this chapter thoroughly.

For detailed functions of the robot operation, read the relevant operator's manual to understand fully its specification.

For the safety of the operator and the system, follow all safety precautions when operating a robot and its peripheral equipment installed in a work cell. For safe use of FANUC robots, you must read and follow the instructions in the *FANUC Robot series SAFETY HANDBOOK (B-80687EN)*.

## PERSONNEL

---

Personnel can be classified as follows:

Operator:

- Turns the robot controller power ON/OFF
- Starts the robot program from the operator panel

Programmer or Teaching operator:

- Operates the robot
- Teaches the robot inside the safeguarded space

Maintenance technician:

- Operates the robot
- Teaches the robot inside the safeguarded space
- Performs maintenance (repair, adjustment, replacement)
- The operator is not allowed to work in the safeguarded space.
- The programmer or teaching operator and maintenance technician are allowed to work in the safeguarded space. Work carried out in the safeguarded space include transportation, installation, teaching, adjustment, and maintenance.
- To work inside the safeguarded space, the person must be trained on proper robot operation.

**Table s-2** lists the work outside the safeguarded space. In this table, the symbol “○” means the work is allowed to be carried out by the specified personnel.

**Table s-2 Work Performed Outside the Safeguarded Space**

	Operator	Programmer or Teaching Operator	Maintenance Technician
Turn power ON/OFF to Robot controller	○	○	○
Select operating mode (AUTO, T1, T2)		○	○
Select remote/local mode		○	○
Select robot program with teach pendant		○	○
Select robot program with external device		○	○
Start robot program with operator's panel	○	○	○

	Operator	Programmer or Teaching Operator	Maintenance Technician
Start robot program with teach pendant		<input type="radio"/>	<input type="radio"/>
Reset alarm with operator's panel		<input type="radio"/>	<input type="radio"/>
Reset alarm with teach pendant		<input type="radio"/>	<input type="radio"/>
Set data on teach pendant		<input type="radio"/>	<input type="radio"/>
Teaching with teach pendant		<input type="radio"/>	<input type="radio"/>
Emergency stop with operator's panel	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Emergency stop with teach pendant	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Operator panel maintenance			<input type="radio"/>
Teach pendant maintenance			<input type="radio"/>

During robot operation, programming and maintenance, the operator, programmer, teaching operator and maintenance engineer take care of their own safety using at least the following safety protectors:

- Use clothes, uniform, overall adequate for the work
- Safety shoes
- Helmet

## DEFINITION OF SAFETY NOTATIONS

To ensure the safety of users and prevent damage to the machine, this manual indicates each precaution on safety with "WARNING" or "CAUTION" according to its severity. Supplementary information is indicated by "NOTE". Read the contents of each "WARNING", "CAUTION" and "NOTE" before using the robot.

Symbol	Definitions
 <b>WARNING</b>	Used if hazard resulting in the death or serious injury of the user will be expected to occur if he or she fails to follow the approved procedure.
 <b>CAUTION</b>	Used if a hazard resulting in the minor or moderate injury of the user, or equipment damage may be expected to occur if he or she fails to follow the approved procedure.
<b>NOTE</b>	Used if a supplementary explanation not related to any of WARNING and CAUTION is to be indicated.

# SAFETY PRECAUTIONS FOR PAINT ROBOTS

Today's highly automated paint booths require that process and maintenance personnel have full awareness of the system and its capabilities. They must understand the interaction that occurs between any moving machinery along the conveyor and the robot(s), hood/deck and door opening devices, and high-voltage electrostatic tools.

Whenever personnel are working inside the paint booth, ventilation equipment must be used. Instruction on the proper use of ventilating equipment is usually provided by the paint shop supervisor.

## **WARNING**

Ensure that all ground cables remain connected. Never operate the paint robot with ground provisions disconnected. Otherwise, you could injure personnel or damage equipment.

Paint robots have the following operation modes:

- Teach or manual mode
- Automatic mode, including automatic or Production mode

During both teach and automatic modes, the robots in the paint booth will follow a predetermined pattern of movements. In teach mode, the process technician teaches (programs) paint paths using the teach pendant.

In automatic mode, robot operation is operated at full production speed. It is initiated at the System Control Console (SCC) or Manual Control Panel (MCP), if available, and can be monitored from outside the paint booth. All personnel must remain outside of the booth whenever automatic mode is initiated.

## Paint System Safety Features

Process technicians and maintenance personnel must become familiar with the equipment and its capabilities. To minimize the risk of injury when working near robots and related equipment, personnel must comply strictly with the procedures in the manuals.

In addition to other safety features, the paint system may include the following:

- Warning beacons
- Intrinsically Safe Teach Pendant (for use in a hazardous area)
- Light curtains, door switches etc.
- Safety I/O

## Staying Safe While Operating the Paint Robot

When you work in or near the paint booth, observe the following rules, in addition to all rules for safe operation that apply to all robot systems.

- Know the work area of the entire paint station.
- Know the work envelope of the robot and hood/deck and door opening devices.
- Know the location and status of all switches, sensors, and/or control signals that might cause the robot, conveyor, and opening devices to move.

- Remove all metallic objects, such as rings, watches, and belts, before entering a booth when the electrostatic devices are enabled.
- Be aware of signals and/or operations that could start up paint process application equipment such as guns or bells.
- Be aware of potential air pressure buildup behind paint robot access covers.
- Be aware of all safety precautions when dispensing of paint is required.

 **WARNING**

1. Never bypass, modify, or otherwise deactivate the robot purge system for any reason. Deactivating the purge system might result in an explosion serious injury or death.
2. Purged enclosures shall not be opened unless the area is known to be nonhazardous or all power has been removed from devices within the enclosure. Power cannot be restored because power is restricted by the purge system and cannot be turned on after the enclosure has been opened until all combustible dusts have been removed from the interior of the enclosure and the enclosure has been purged.

## Special Precautions for Combustible Dusts (Powder Paint)

When the robot is used in a location where combustible dusts are found, such as the application of powder paint, the following special precautions are required to insure that there are no combustible dusts inside the robot.

- Purge maintenance air should be maintained at all times, even when the robot power is off. This will insure that dust cannot enter the robot.
- A purge cycle will not remove accumulated dusts. Therefore, if the robot is exposed to dust when maintenance air is not present, it will be necessary to remove the covers and clean out any accumulated dust.
- Do not energize the robot until you have performed the following steps.
  1. Before covers are removed, the exterior of the robot should be cleaned to remove accumulated dust.
  2. When cleaning and removing accumulated dust, either on the outside or inside of the robot, be sure to use methods appropriate for the type of dust that exists. According to the type of dust, wipe off any dust with lint free rags dampened with an appropriate cleaning solvent. Do not use vacuum cleaners to remove dust unless they are designed for use in a hazardous area. Otherwise, it can generate static electricity and cause an explosion.
  3. Thoroughly clean the interior of the robot with a lint free rag to remove any accumulated dust.
  4. When the dust has been removed, the covers must be replaced immediately.
  5. Immediately after the covers are replaced, run a complete purge cycle. The robot can now be energized.

## Staying Safe While Operating Paint Application Equipment

When you work with paint application equipment, observe the following rules, in addition to all rules for safe operation that apply to all robot systems.

**⚠️ WARNING**

When working with electrostatic paint equipment, follow all national and local codes as well as all safety guidelines within your organization.

**Grounding:** All electrically conductive objects in the spray area must be grounded. This includes the spray booth, robots, conveyors, workstations, part carriers, hooks, paint pressure pots, as well as solvent containers.

**High Voltage:** High voltage should only be on during actual spray operations. Voltage should be off when the painting process is completed. Never leave high voltage on during a cap cleaning process.

1. Avoid any accumulation of combustible vapors or coating matter.
2. Follow all manufacturer recommended cleaning procedures.
3. Make sure all interlocks are operational.
4. No smoking.
5. When you perform maintenance on the paint bell, disable all air and paint pressure to the paint bell.
6. Verify that the lines are not under pressure.
7. Perform a risk assessment appropriate for paint application equipment.

## Staying Safe During Maintenance

Paint robots operate in a potentially explosive environment. When you perform maintenance on the paint bell, disable all air and paint pressure to the paint bell.

1. Adequate booth ventilation is required. Overexposure to harmful fumes could cause drowsiness or skin and eye irritation.
2. Wear appropriate Personal Protective Equipment (PPE) for the application.
3. Use tools that are appropriate for hazardous areas.
4. Be sure all covers and inspection plates are in good repair and in place.
5. Whenever possible, turn off the main electrical disconnect before you clean the robot.
6. When using solvents, observe the following:
  - Wear eye protection and protective gloves during application and removal.
  - If there is contact with the skin, wash with water.
  - Follow the Original Equipment Manufacturer's Material Safety Data Sheets.

EFFMAN  
QUIRIONR

# TABLE OF CONTENTS

---



---

<b>SAFETY PRECAUTIONS.....</b>	<b>s-1</b>
<b>SAFETY PRECAUTIONS.....</b>	<b>s-1</b>
<b>1 KAREL LANGUAGE OVERVIEW .....</b>	<b>1</b>
1.1 KAREL PROGRAMMING LANGUAGE .....	1
1.1.1 Creating a Program .....	2
1.1.2 Translating a Program .....	2
1.1.3 Loading Program Logic and Data.....	3
1.1.4 Executing a Program .....	4
1.1.5 Execution History.....	4
1.1.6 Program Structure.....	4
1.2 SYSTEM SOFTWARE.....	5
1.2.1 Software Components.....	5
1.2.2 Supported Robots .....	6
1.3 CONTROLLER .....	6
1.3.1 Memory.....	6
1.3.2 Input/Output System .....	8
1.3.3 User Interface Devices.....	8
1.3.4 Frames.....	9
<b>2 LANGUAGE ELEMENTS .....</b>	<b>11</b>
2.1 LANGUAGE COMPONENTS .....	11
2.1.1 Character Set .....	11
2.1.2 Operators .....	13
2.1.3 Reserved Words .....	13
2.1.4 User-Defined Identifiers .....	14
2.1.5 Labels .....	15
2.1.6 Predefined Identifiers .....	15
2.1.7 System Variables .....	17
2.1.8 Comments .....	17
2.2 TRANSLATOR DIRECTIVES.....	18
2.3 DATA TYPES .....	19
2.4 USER-DEFINED DATA TYPES AND STRUCTURES .....	20
2.4.1 User-Defined Data Types .....	20
2.4.2 User-Defined Data Structures .....	22
2.5 ARRAYS .....	23
2.5.1 Multi-Dimensional Arrays .....	24
2.5.2 Variable-Sized Arrays .....	25
<b>3 USE OF OPERATORS .....</b>	<b>27</b>
3.1 EXPRESSIONS AND ASSIGNMENTS .....	27
3.1.1 Rule for Expressions and Assignments .....	27
3.1.2 Evaluation of Expressions and Assignments .....	27
3.1.3 Variables and Expressions .....	29
3.2 OPERATIONS .....	29
3.2.1 Arithmetic Operations .....	30
3.2.2 Relational Operations .....	31
3.2.3 Boolean Operations .....	32
3.2.4 Special Operations .....	32

<b>4 PROGRAM CONTROL .....</b>	<b>37</b>
4.1 PROGRAM CONTROL STRUCTURES .....	37
4.1.1 Alteration Control Structures .....	37
4.1.2 Looping Control Statements .....	37
4.1.3 Unconditional Branch Statement .....	38
4.1.4 Execution Control Statements .....	38
4.1.5 Condition Handlers .....	38
<b>5 ROUTINES .....</b>	<b>39</b>
5.1 ROUTINE EXECUTION .....	39
5.1.1 Declaring Routines .....	39
5.1.2 Invoking Routines .....	41
5.1.3 Returning from Routines .....	42
5.1.4 Scope of Variables .....	44
5.1.5 Parameters and Arguments .....	45
5.1.6 Stack Usage .....	48
5.2 BUILT- IN ROUTINES .....	50
<b>6 CONDITION HANDLERS .....</b>	<b>55</b>
6.1 CONDITION HANDLER OPERATIONS .....	56
6.1.1 Global Condition Handlers .....	56
6.2 CONDITIONS .....	58
6.2.1 Port_Id Conditions .....	59
6.2.2 Relational Conditions .....	59
6.2.3 System and Program Event Conditions .....	60
6.3 ACTIONS .....	62
6.3.1 Assignment Actions .....	62
6.3.2 Motion Related Actions .....	63
6.3.3 Routine Call Actions .....	64
6.3.4 Miscellaneous Actions .....	64
<b>7 FILE INPUT/OUTPUT OPERATIONS .....</b>	<b>67</b>
7.1 FILE VARIABLES .....	67
7.2 OPEN FILE STATEMENT .....	68
7.2.1 Setting File and Port Attributes .....	68
7.2.2 File String .....	73
7.2.3 Usage String .....	74
7.3 CLOSE FILE STATEMENT .....	76
7.4 READ STATEMENT .....	76
7.5 WRITE STATEMENT .....	78
7.6 INPUT/OUTPUT BUFFER .....	78
7.7 FORMATTING TEXT (ASCII) INPUT/OUTPUT .....	79
7.7.1 Formatting INTEGER Data Items .....	80
7.7.2 Formatting REAL Data Items .....	82
7.7.3 Formatting BOOLEAN Data Items .....	83
7.7.4 Formatting STRING Data Items .....	85
7.7.5 Formatting VECTOR Data Items .....	87
7.7.6 Formatting Positional Data Items .....	87
7.8 FORMATTING BINARY INPUT/OUTPUT .....	88
7.8.1 Formatting INTEGER Data Items .....	89
7.8.2 Formatting REAL Data Items .....	90
7.8.3 Formatting BOOLEAN Data Items .....	90

7.8.4 Formatting STRING Data Items .....	90
7.8.5 Formatting VECTOR Data Items .....	91
7.8.6 Formatting POSITION Data Items .....	91
7.8.7 Formatting XYZWPR Data Items .....	91
7.8.8 Formatting XYZWPREXT Data Items .....	92
7.8.9 Formatting JOINTPOS Data Items .....	92
<b>7.9 USER INTERFACE TIPS .....</b>	<b>92</b>
7.9.1 USER Menu on the Teach Pendant .....	92
7.9.2 USER Menu on the CRT/KB .....	93
<b>8 POSITION DATA .....</b>	<b>95</b>
8.1 POSITIONAL DATA .....	95
8.2 FRAMES OF REFERENCE .....	96
8.2.1 World Frame .....	97
8.2.2 User Frame (UFRAME) .....	97
8.2.3 Tool Definition (UTOOL) .....	97
8.2.4 Using Frames in the Teach Pendant Editor (TP) .....	98
8.3 JOG COORDINATE SYSTEMS .....	98
<b>9 TEACHING KAREL VARIABLES .....</b>	<b>101</b>
9.1 KAREL Positions .....	101
9.1.1 Teaching KAREL Positions .....	101
9.2 KAREL Paths .....	104
9.2.1 Teaching KAREL Paths .....	105
9.3 KAREL Variables .....	111
9.3.1 Modifying KAREL Variables .....	111
<b>10 FILE SYSTEM .....</b>	<b>113</b>
10.1 FILE SPECIFICATION .....	113
10.1.1 Device Name .....	114
10.1.2 File Name .....	115
10.1.3 File Type .....	116
10.2 STORAGE DEVICE ACCESS .....	117
10.2.1 Storage Device Access Overview .....	117
10.2.2 Memory File Devices .....	122
10.2.3 Virtual Devices .....	123
10.2.4 File Pipes .....	124
10.3 FILE ACCESS .....	128
10.4 FORMATTING XML INPUT .....	128
10.4.1 Installation Sequence .....	129
10.4.2 Example KAREL Program Referencing an XML File .....	129
10.4.3 Parse Errors .....	133
10.5 MEMORY DEVICE .....	133
<b>11 DICTIONARIES AND FORMS .....</b>	<b>137</b>
11.1 CREATING USER DICTIONARIES .....	137
11.1.1 Dictionary Syntax .....	137
11.1.2 Dictionary Element Number .....	138
11.1.3 Dictionary Element Name .....	138
11.1.4 Dictionary Cursor Positioning .....	139
11.1.5 Dictionary Element Text .....	139
11.1.6 Dictionary Reserved Word Commands .....	141
11.1.7 Character Codes .....	143

11.1.8 Nesting Dictionary Elements .....	143
11.1.9 Dictionary Comment .....	143
11.1.10 Generating a KAREL Constant File .....	143
11.1.11 Compressing and Loading Dictionaries on the Controller .....	144
11.1.12 Accessing Dictionary Elements from a KAREL Program .....	145
<b>11.2 CREATING USER FORMS .....</b>	<b>145</b>
11.2.1 Form Syntax .....	146
11.2.2 Form Attributes .....	148
11.2.3 Form Title and Menu Label .....	148
11.2.4 Form Menu Text .....	149
11.2.5 Form Selectable Menu Item .....	150
11.2.6 Edit Data Item .....	151
11.2.7 Dynamic Forms using Tree View.....	157
11.2.8 Non-Selectable Text .....	157
11.2.9 Display Only Data Items .....	157
11.2.10 Cursor Position Attributes .....	157
11.2.11 Form Reserved Words and Character Codes .....	158
11.2.12 Form Function Key Element Name or Number .....	159
11.2.13 Form Function Key Using a Variable .....	160
11.2.14 Form Help Element Name or Number .....	161
11.2.15 Teach Pendant Form Screen .....	161
11.2.16 CRT/KB Form Screen .....	162
11.2.17 Form File Naming Convention .....	162
11.2.18 Compressing and Loading Forms on the Controller .....	163
11.2.19 Displaying a Form .....	164
<b>12 SOCKET MESSAGING .....</b>	<b>173</b>
<b>12.1 SYSTEM REQUIREMENTS .....</b>	<b>173</b>
12.1.1 Software Requirements .....	173
12.1.2 Hardware Requirements .....	173
<b>12.2 CONFIGURING THE SOCKET MESSAGING OPTION .....</b>	<b>173</b>
12.2.1 Setting up a Server Tag .....	174
12.2.1.1 Setting up a Server Tag .....	174
12.2.2 Setting up a Client Tag .....	176
12.2.2.1 Setting up a ClientTag .....	176
<b>12.3 SOCKET MESSAGING AND KAREL .....</b>	<b>178</b>
12.3.1 MSG_CONNECT(string, integer) .....	179
12.3.2 MSG_DISCO(string, integer) .....	179
12.3.3 MSG_PING(string, integer) .....	179
12.3.4 Exchanging Data during a Socket Messaging Connection .....	180
<b>12.4 NETWORK PERFORMANCE .....</b>	<b>180</b>
12.4.1 Guidelines for a Good Implementation .....	180
<b>12.5 PROGRAMMING EXAMPLES .....</b>	<b>180</b>
12.5.1 A KAREL Client Application .....	181
12.5.2 A KAREL Server Application .....	183
12.5.3 ANSI C Loopback Client Example .....	184
<b>13 DATA TRANSFER BETWEEN ROBOTS OVER ETHERNET (DTBR)..</b>	<b>187</b>
<b>13.1 TERMINOLOGY.....</b>	<b>187</b>
<b>13.2 SETUP.....</b>	<b>187</b>
13.2.1 TCP/IP Setup.....	188
<b>13.3 TCP/IP SETUP FOR ROBOGUIDE.....</b>	<b>189</b>
<b>13.4 STANDARD DATA TRANSFER PROGRAMS.....</b>	<b>190</b>

13.4.1 RGETNREG: Program to Get Numeric Register.....	191
13.4.2 RSETNREG: Program to Set Numeric Register.....	192
13.4.3 RGETPREG: Program to Get Position Register.....	192
13.4.4 RSETPREG: Program to Set Position Register.....	194
<b>13.5 ERROR RECOVERY.....</b>	<b>195</b>
13.6 KAREL BUILT-INS.....	196
13.7 TIME OUT AND RETRY.....	197
13.8 LIMITATIONS.....	197
13.9 TROUBLESHOOTING.....	198
<b>14 SYSTEM VARIABLES .....</b>	<b>201</b>
14.1 ACCESS RIGHTS .....	201
14.2 STORAGE .....	201
<b>15 KAREL COMMAND LANGUAGE (KCL) .....</b>	<b>203</b>
15.1 COMMAND FORMAT .....	203
15.1.1 Default Program .....	203
15.1.2 Variables and Data Types .....	203
15.2 PROGRAM CONTROL COMMANDS .....	204
15.3 ENTERING COMMANDS .....	204
15.3.1 Abbreviations .....	205
15.3.2 Error Messages .....	205
15.3.3 Subdirectories .....	205
15.4 COMMAND PROCEDURES .....	205
15.4.1 Command Procedure Format .....	205
15.4.2 Creating Command Procedures .....	206
15.4.3 Error Processing .....	206
15.4.4 Executing Command Procedures .....	206
<b>16 INPUT/OUTPUT SYSTEM .....</b>	<b>209</b>
16.1 USER-DEFINED SIGNALS .....	209
16.1.1 DIN and DOUT Signals .....	209
16.1.2 GIN and GOUT Signals .....	210
16.1.3 AIN and AOUT Signals .....	210
16.1.4 Hand Signals .....	211
16.2 SYSTEM-DEFINED SIGNALS .....	212
16.2.1 Robot Digital Input and Output Signals (RDI/RDO) .....	212
16.2.2 Operator Panel Input and Output Signals (OPIN/OPOUT) .....	212
16.2.3 Teach Pendant Input and Output Signals (TPIN/TPOUT) .....	219
16.3 SERIAL INPUT/OUTPUT .....	225
<b>17 MULTI-TASKING .....</b>	<b>229</b>
17.1 MULTI-TASKING TERMINOLOGY .....	229
17.2 INTERPRETER ASSIGNMENT .....	230
17.3 MOTION CONTROL .....	230
17.4 TASK SCHEDULING .....	231
17.4.1 Priority Scheduling .....	231
17.4.2 Time Slicing .....	232
17.5 STARTING TASKS .....	232
17.5.1 Running Programs from the User Operator Panel (UOP) PNS Signal .....	233
17.5.2 Child Tasks .....	233
17.6 TASK CONTROL AND MONITORING .....	233
17.6.1 From TPP Programs .....	233

17.6.2 From KAREL Programs .....	234
17.6.3 From KCL .....	234
<b>17.7 USING SEMAPHORES AND TASK SYNCHRONIZATION .....</b>	<b>234</b>
<b>17.8 USING QUEUES FOR TASK COMMUNICATIONS .....</b>	<b>239</b>

## APPENDIX

### A KAREL LANGUAGE ALPHABETICAL DESCRIPTION ..... 243

<b>A.1 - A - KAREL LANGUAGE DESCRIPTION .....</b>	<b>250</b>
A.1.1 ABORT Action .....	250
A.1.2 ABORT Condition .....	251
A.1.3 ABORT Statement .....	251
A.1.4 ABORT_TASK Built-In Procedure .....	252
A.1.5 ABS Built-In Function .....	252
A.1.6 ACOS Built-In Function .....	252
A.1.7 ACT_SCREEN Built-In Procedure .....	253
A.1.8 ACT_TBL Built-In Procedure.....	254
A.1.9 ADD_BYNAMEPC Built-In Procedure .....	255
A.1.10 ADD_DICT Built-In Procedure .....	256
A.1.11 ADD_INTPC Built-In Procedure .....	257
A.1.12 ADD_REALPC Built-In Procedure .....	258
A.1.13 ADD_STRINGPC Built-In Procedure .....	259
A.1.14 %ALPHABETIZE Translator Directive .....	260
A.1.15 APPEND_NODE Built-In Procedure .....	260
A.1.16 APPEND_QUEUE Built-In Procedure .....	261
A.1.17 APPROACH Built-In Function .....	261
A.1.18 ARRAY Data Type .....	262
A.1.19 ARRAY_LEN Built-In Function .....	263
A.1.20 ASIN Built-In Function .....	263
A.1.21 Assignment Action .....	264
A.1.22 Assignment Statement .....	265
A.1.23 ATAN2 Built-In Function .....	266
A.1.24 ATTACH Statement .....	267
A.1.25 ATT_WINDOW_D Built-In Procedure .....	267
A.1.26 ATT_WINDOW_S Built-In Procedure .....	268
A.1.27 AVL_POS_NUM Built-In Procedure .....	269
<b>A.2 - B - KAREL LANGUAGE DESCRIPTION .....</b>	<b>269</b>
A.2.1 BOOLEAN Data Type .....	269
A.2.2 BYNAME Built-In Function .....	270
A.2.3 BYTE Data Type .....	271
A.2.4 BYTES_AHEAD Built-In Procedure .....	271
A.2.5 BYTES_LEFT Built-In Function .....	272
<b>A.3 - C - KAREL LANGUAGE DESCRIPTION .....</b>	<b>274</b>
A.3.1 CALL_PROG Built-In Procedure .....	274
A.3.2 CALL_PROGLIN Built-In Procedure .....	274
A.3.3 CANCEL Action .....	275
A.3.4 CANCEL Statement .....	275
A.3.5 CANCEL_FILE Statement .....	276
A.3.6 CHECK_DICT Built-In Procedure .....	277
A.3.7 CHECK_EPOS Built-In Procedure .....	277
A.3.8 CHECK_NAME Built-In Procedure .....	278
A.3.9 CHR Built-In Function .....	278

A.3.10 CLEAR Built-In Procedure .....	279
A.3.11 CLEAR_SEMA Built-In Procedure .....	279
A.3.12 CLOSE FILE Statement .....	280
A.3.13 CLOSE HAND Statement .....	280
A.3.14 CLOSE_TPE Built-In Procedure .....	281
A.3.15 CLR_IO_STAT Built-In Procedure .....	281
A.3.16 CLR_PORT_SIM Built-In Procedure .....	282
A.3.17 CLR_POS_REG Built-In Procedure .....	282
A.3.18 %CMOSVARS Translator Directive .....	283
A.3.19 %CMOS2SHADOW Translator Directive .....	283
A.3.20 CNC_DYN_DISB Built-In Procedure .....	283
A.3.21 CNC_DYN_DISE Built-In Procedure .....	284
A.3.22 CNC_DYN_DISI Built-In Procedure .....	284
A.3.23 CNC_DYN_DISP Built-In Procedure .....	285
A.3.24 CNC_DYN_DISR Built-In Procedure .....	285
A.3.25 CNC_DYN_DISS Built-In Procedure .....	286
A.3.26 CNCL_STP_MTN Built-In Procedure .....	286
A.3.27 CNV_CNF_STRG Built-In Procedure .....	287
A.3.28 CNV_CONF_STR Built-In Procedure .....	288
A.3.29 CNV_INT_STR Built-In Procedure .....	288
A.3.30 CNV_JPOS_REL Built-In Procedure .....	289
A.3.31 CNV_REAL_STR Built-In Procedure .....	289
A.3.32 CNV_REL_JPOS Built-In Procedure .....	290
A.3.33 CNV_STR_CONF Built-In Procedure .....	291
A.3.34 CNV_STR_INT Built-In Procedure .....	291
A.3.35 CNV_STR_REAL Built-In Procedure .....	292
A.3.36 CNV_STR_TIME Built-In Procedure .....	292
A.3.37 CNV_TIME_STR Built-In Procedure .....	293
A.3.38 %COMMENT Translator Directive .....	293
A.3.39 COMPARE_FILE Built-in Procedure .....	294
A.3.40 CONDITION...ENDCONDITION Statement .....	296
A.3.41 CONFIG Data Type .....	297
A.3.42 CONNECT TIMER Statement .....	298
A.3.43 CONTINUE Action .....	298
A.3.44 CONTINUE Condition .....	299
A.3.45 CONT_TASK Built-In Procedure .....	299
A.3.46 COPY_FILE Built-In Procedure .....	300
A.3.47 COPY_PATH Built-In Procedure .....	301
A.3.48 COPY_QUEUE Built-In Procedure .....	302
A.3.49 COPY_TPE Built-In Procedure .....	303
A.3.50 COS Built-In Function .....	304
A.3.51 CR Input/Output Item .....	304
A.3.52 CREATE_TPE Built-In Procedure .....	305
A.3.53 CREATE_VAR Built-In Procedure .....	306
A.3.54 %CRTDEVICE Translator Directive .....	307
A.3.55 CURJPOS Built-In Function .....	308
A.3.56 CURPOS Built-In Function .....	308
A.3.57 CURR_PROG Built-In Function .....	309
<b>A.4 - D - KAREL LANGUAGE DESCRIPTION .....</b>	<b>309</b>
A.4.1 DAQ_CHECKP Built-In Procedure .....	310
A.4.2 DAQ_REGPIPE Built-In Procedure .....	310
A.4.3 DAQ_START Built-In Procedure .....	312
A.4.4 DAQ_STOP Built-In Procedure .....	313

A.4.5 DAQ_UNREG Built-In Procedure .....	314
A.4.6 DAQ_WRITE Built-In Procedure .....	315
A.4.7 %DEFGROUP Translator Directive .....	316
A.4.8 DEF_SCREEN Built-In Procedure .....	316
A.4.9 DEF_WINDOW Built-In Procedure .....	317
A.4.10 %DELAY Translator Directive .....	318
A.4.11 DELAY Statement .....	318
A.4.12 DELETE_FILE Built-In Procedure .....	319
A.4.13 DELETE_NODE Built-In Procedure .....	320
A.4.14 DELETE_QUEUE Built-In Procedure .....	320
A.4.15 DEL_INST_TPE Built-In Procedure .....	321
A.4.16 DET_WINDOW Built-In Procedure .....	321
A.4.17 DISABLE CONDITION Action .....	322
A.4.18 DISABLE CONDITION Statement .....	322
A.4.19 DISCONNECT TIMER Statement .....	323
A.4.20 DISCTRL_ALPH Built-In Procedure .....	324
A.4.21 DISCTRL_FORM Built-In Procedure .....	325
A.4.22 DISCTRL_LIST Built-In Procedure .....	327
A.4.23 DISCTRL_PLMN Built-In Procedure .....	328
A.4.24 DISCTRL_SBMN Built-In Procedure .....	329
A.4.25 DISCTRL_TBL Built-In Procedure .....	331
A.4.26 DISMOUNT_DEV Built-In Procedure .....	333
A.4.27 DISP_DAT_T Data Type .....	333
A.4.28 DOSFILE_INF Built-In Procedure.....	335
<b>A.5 - E - KAREL LANGUAGE DESCRIPTION .....</b>	<b>336</b>
A.5.1 ENABLE CONDITION Action .....	336
A.5.2 ENABLE CONDITION Statement .....	336
A.5.3 %ENVIRONMENT Translator Directive .....	337
A.5.4 ERR_DATA Built-In Procedure .....	338
A.5.5 ERROR Condition .....	339
A.5.6 EVAL Clause .....	340
A.5.7 EVENT Condition .....	340
A.5.8 EXP Built-In Function .....	341
<b>A.6 - F - KAREL LANGUAGE DESCRIPTION .....</b>	<b>341</b>
A.6.1 FILE Data Type .....	341
A.6.2 FILE_LIST Built-In Procedure .....	342
A.6.3 FOR...ENDFOR Statement .....	343
A.6.4 FORCE_LINK Built-In Procedure.....	344
A.6.5 FORCE_SPMENU Built-In Procedure .....	345
A.6.6 FORMAT_DEV Built-In Procedure .....	347
A.6.7 FRAME Built-In Function .....	348
A.6.8 FROM Clause .....	349
<b>A.7 - G - KAREL LANGUAGE DESCRIPTION .....</b>	<b>349</b>
A.7.1 GET_ATTR_PRG Built-In Procedure .....	349
A.7.2 GET_FILE_POS Built-In Function .....	351
A.7.3 GET_JPOS_REG Built-In Function .....	352
A.7.4 GET_JPOS_TPE Built-In Function .....	352
A.7.5 GET_PORT_ASG Built-in Procedure .....	353
A.7.6 GET_PORT_ATR Built-In Function .....	354
A.7.7 GET_PORT_CMT Built-In Procedure .....	356
A.7.8 GET_PORT_MOD Built-In Procedure .....	356
A.7.9 GET_PORT_SIM Built-In Procedure .....	358
A.7.10 GET_PORT_VAL Built-In Procedure .....	358

A.7.11 GET_POS_FRM Built-In Procedure .....	359
A.7.12 GET_POS_REG Built-In Function .....	359
A.7.13 GET_POS_TPE Built-In Function .....	360
A.7.14 GET_POS_TYP Built-In Procedure .....	361
A.7.15 GET_PREG_CMT Built-In-Procedure .....	361
A.7.16 GET_QUEUE Built-In Procedure .....	362
A.7.17 GET_REG Built-In Procedure .....	363
A.7.18 GET_REG_CMT Built-In Procedure.....	363
A.7.19 GET_SREG_CMT Built-In Procedure.....	364
A.7.20 GET_STR_REG Built-In Procedure.....	364
A.7.21 GET_TIME Built-In Procedure .....	365
A.7.22 GET_TPE_CMT Built-in Procedure .....	365
A.7.23 GET_TPE_PRM Built-in Procedure .....	366
A.7.24 GET_TSK_INFO Built-In Procedure .....	368
A.7.25 GET_USEC_SUB Built-In Procedure .....	369
A.7.26 GET_USEC_TIM Built-In Function .....	369
A.7.27 GET_VAR Built-In Procedure .....	370
A.7.28 GO TO Statement .....	373
<b>A.8 - H - KAREL LANGUAGE DESCRIPTION .....</b>	<b>374</b>
A.8.1 HOLD Action .....	374
A.8.2 HOLD Statement .....	374
<b>A.9 - I - KAREL LANGUAGE DESCRIPTION .....</b>	<b>375</b>
A.9.1 IF ... ENDIF Statement .....	375
A.9.2 IN Clause .....	376
A.9.3 %INCLUDE Translator Directive .....	376
A.9.4 INDEX Built-In Function .....	377
A.9.5INI_DYN_DISB Built-In Procedure .....	378
A.9.6INI_DYN_DISE Built-In Procedure .....	379
A.9.7INI_DYN_DISI Built-In Procedure .....	380
A.9.8INI_DYN_DISP Built-In Procedure .....	381
A.9.9INI_DYN_DISR Built-In Procedure .....	382
A.9.10INI_DYN_DISS Built-In Procedure .....	383
A.9.11INIT_QUEUE Built-In Procedure .....	384
A.9.12INIT_TBL Built-In Procedure .....	384
A.9.13IN_RANGE Built-In Function .....	393
A.9.14INSERT_NODE Built-In Procedure .....	393
A.9.15INSERT_QUEUE Built-In Procedure .....	394
A.9.16INTEGER Data Type .....	395
A.9.17INV Built-In Function .....	396
A.9.18IO_MOD_TYPE Built-In Procedure .....	396
A.9.19IO_STATUS Built-In Function .....	397
<b>A.10 - J - KAREL LANGUAGE DESCRIPTION .....</b>	<b>398</b>
A.10.1J_IN_RANGE Built-In Function .....	398
A.10.2JOINTPOS Data Type .....	399
A.10.3JOINT2POS Built-In Function .....	399
<b>A.11 - K - KAREL LANGUAGE DESCRIPTION .....</b>	<b>400</b>
A.11.1KCL Built-In Procedure .....	400
A.11.2KCL_NO_WAIT Built-In Procedure .....	401
A.11.3KCL_STATUS Built-In Procedure .....	402
<b>A.12 - L - KAREL LANGUAGE DESCRIPTION .....</b>	<b>402</b>
A.12.1LN Built-In Function .....	402
A.12.2LOAD Built-In Procedure .....	403
A.12.3LOAD_STATUS Built-In Procedure .....	404

A.12.4 LOCK_GROUP Built-In Procedure .....	404
A.12.5 %LOCKGROUP Translator Directive .....	405
<b>A.13 - M - KAREL LANGUAGE DESCRIPTION .....</b>	<b>406</b>
A.13.1 MIRROR Built-In Function .....	406
A.13.2 MODIFY_QUEUE Built-In Procedure .....	407
A.13.3 MOTION_CTL Built-In Function .....	408
A.13.4 MOUNT_DEV Built-In Procedure .....	409
A.13.5 MOVE_FILE Built-In Procedure .....	409
A.13.6 MSG_CONNECT Built-In Procedure .....	410
A.13.7 MSG_DISCO Built-In Procedure .....	411
A.13.8 MSG_PING Built-In Procedure .....	412
<b>A.14 - N - KAREL LANGUAGE DESCRIPTION .....</b>	<b>412</b>
A.14.1 NOABORT Action .....	412
A.14.2 %NOABORT Translator Directive .....	413
A.14.3 %NOBUSYLAMP Translator Directive .....	413
A.14.4 NODE_SIZE Built-In Function .....	413
A.14.5 %NOLOCKGROUP Translator Directive .....	415
A.14.6 NOMESSAGE Action .....	415
A.14.7 NOPAUSE Action .....	416
A.14.8 %NOPAUSE Translator Directive .....	416
A.14.9 %NOPAUSESHFT Translator Directive .....	417
<b>A.15 - O - KAREL LANGUAGE DESCRIPTION .....</b>	<b>417</b>
A.15.1 OPEN FILE Statement .....	417
A.15.2 OPEN HAND Statement .....	418
A.15.3 OPEN_TPE Built-In Procedure .....	418
A.15.4 ORD Built-In Function .....	419
A.15.5 ORIENT Built-In Function .....	420
<b>A.16 - P - KAREL LANGUAGE DESCRIPTION .....</b>	<b>420</b>
A.16.1 PATH Data Type .....	420
A.16.2 PATH_LEN Built-In Function .....	422
A.16.3 PAUSE Action .....	422
A.16.4 PAUSE Condition .....	423
A.16.5 PAUSE Statement .....	423
A.16.6 PAUSE_TASK Built-In Procedure .....	424
A.16.7 PEND_SEMA Built-In Procedure .....	425
A.16.8 PIPE_CONFIG Built-In Procedure .....	426
A.16.9 POP_KEY_RD Built-In Procedure .....	426
A.16.10 Port_Id Action .....	427
A.16.11 Port_Id Condition .....	427
A.16.12 POS Built-In Function .....	428
A.16.13 POS2JOINT Built-In Function .....	429
A.16.14 POS_REG_TYPE Built-In Procedure .....	430
A.16.15 POSITION Data Type .....	431
A.16.16 POST_ERR Built-In Procedure .....	432
A.16.17 POST_ERR_L Built-In Procedure .....	432
A.16.18 POST_SEMA Built-In Procedure .....	433
A.16.19 PRINT_FILE Built-In Procedure .....	434
A.16.20 %PRIORITY Translator Directive .....	434
A.16.21 PROG_BACKUP Built-In Procedure .....	435
A.16.22 PROG_CLEAR Built-In Procedure .....	436
A.16.23 PROG_LIST Built-In Procedure .....	438
A.16.24 PROG_RESTORE Built-In Procedure .....	439
A.16.25 PROGRAM Statement .....	440

A.16.26 PULSE Action .....	441
A.16.27 PULSE Statement .....	441
A.16.28 PURGE CONDITION Statement .....	442
A.16.29 PURGE_DEV Built-In Procedure .....	443
A.16.30 PUSH_KEY_RD Built-In Procedure .....	443
<b>A.17 - Q - KAREL LANGUAGE DESCRIPTION .....</b>	<b>444</b>
A.17.1 QUEUE_ATTACH Built-in Procedure.....	444
A.17.2 QUEUE_TYPE Data Type .....	446
<b>A.18 - R - KAREL LANGUAGE DESCRIPTION .....</b>	<b>446</b>
A.18.1 READ Statement .....	446
A.18.2 READ_DICT Built-In Procedure .....	447
A.18.3 READ_DICT_V Built-In-Procedure .....	448
A.18.4 READ_KB Built-In Procedure .....	449
A.18.5 REAL Data Type .....	452
A.18.6 Relational Condition .....	453
A.18.7 RELAX HAND Statement .....	454
A.18.8 RELEASE Statement .....	455
A.18.9 REMOVE_DICT Built-In Procedure .....	455
A.18.10 RENAME_FILE Built-In Procedure .....	456
A.18.11 RENAME_VAR Built-In Procedure .....	456
A.18.12 RENAME_VARS Built-In Procedure .....	457
A.18.13 REPEAT ... UNTIL Statement .....	457
A.18.14 RESET Built-In Procedure .....	458
A.18.15 RESUME Action .....	458
A.18.16 RESUME Statement .....	459
A.18.17 RETURN Statement .....	460
A.18.18 RGET_PORTCMT Built-In Routine.....	460
A.18.19 RGET_PORTSIM Built-In Routine.....	461
A.18.20 RGET_PORTVAL Built-In Routine.....	462
A.18.21 RGET_PREGCMT Built-In Routine.....	462
A.18.22 RGET_REG Built-In Routine.....	463
A.18.23 RGET_REG_CMT Built-In Routine.....	464
A.18.24 RGET_SREGCMT Built-in Routine.....	465
A.18.25 RGET_STR_REG Built-In Routine.....	465
A.18.26 RMCN_ALERT Built-In Routine.....	466
A.18.27 RMCN_SEND Built-in Routine.....	467
A.18.28 RNUMREG_RECV Built-In Routine.....	468
A.18.29 RNUMREG_SEND Built-In Routine.....	469
A.18.30 ROUND Built-In Function .....	470
A.18.31 ROUTINE Statement .....	470
A.18.32 RPREG_RECV Built-In Routine.....	471
A.18.33 RPREG_SEND Built-in Routine.....	472
A.18.34 RSET_INT_REG Built-in Routine.....	473
A.18.35 RSET_PORTCMT Built-in Routine.....	474
A.18.36 RSET_PORTSIM Built-in Routine.....	475
A.18.37 RSET_PORTVAL Built-in Routine.....	475
A.18.38 RSET_PREGCMT Built-in Routine.....	476
A.18.39 RSET_REALREG Built-in Routine.....	477
A.18.40 RSET_REG_CMT Built-In Routine.....	477
A.18.41 RSET_SREGCMT Built-in Routine.....	478
A.18.42 RSET_STR_REG Built-in Routine.....	479
A.18.43 RUN_TASK Built-In Procedure .....	479
<b>A.19 - S - KAREL LANGUAGE DESCRIPTION .....</b>	<b>480</b>

A.19.1 SAVE Built-In Procedure .....	481
A.19.2 SAVE_DRAM Built-In Procedure .....	481
A.19.3 SELECT ... ENDSELECT Statement .....	482
A.19.4 SELECT_TPE Built-In Procedure .....	482
A.19.5 SEMA_COUNT Built-In Function .....	483
A.19.6 SEMAPHORE Condition .....	483
A.19.7 SEND_DATAPC Built-In Procedure .....	484
A.19.8 SEND_EVENTPC Built-In Procedure .....	485
A.19.9 SET_ATTR_PRG Built-In Procedure .....	485
A.19.10 SET_CURSOR Built-In Procedure .....	486
A.19.11 SET_EPOS_REG Built-In Procedure .....	487
A.19.12 SET_EPOS_TPE Built-In Procedure .....	488
A.19.13 SET_FILE_ATR Built-In Procedure .....	489
A.19.14 SET_FILE_POS Built-In Procedure .....	489
A.19.15 SET_INT_REG Built-In Procedure .....	490
A.19.16 SET_JPOS_REG Built-In Procedure .....	491
A.19.17 SET_JPOS_TPE Built-In Procedure .....	491
A.19.18 SET_LANG Built-In Procedure .....	492
A.19.19 SET_PERCH Built-In Procedure .....	493
A.19.20 SET_PORT_ASG Built-In Procedure .....	493
A.19.21 SET_PORT_ATR Built-In Function .....	494
A.19.22 SET_PORT_CMT Built-In Procedure .....	496
A.19.23 SET_PORT_MOD Built-In Procedure .....	497
A.19.24 SET_PORT_SIM Built-In Procedure .....	497
A.19.25 SET_PORT_VAL Built-In Procedure .....	498
A.19.26 SET_POS_REG Built-In Procedure .....	499
A.19.27 SET_POS_TPE Built-In Procedure .....	500
A.19.28 SET_PREG_CMT Built-In-Procedure .....	500
A.19.29 SET_REAL_REG Built-In Procedure .....	501
A.19.30 SET_REG_CMT Built-In-Procedure .....	501
A.19.31 SET_SREG_CMT Built-in Procedure .....	501
A.19.32 SET_STR_REG Built-in Procedure .....	502
A.19.33 SET_TIME Built-In Procedure .....	502
A.19.34 SET_TPE_CMT Built-In Procedure .....	503
A.19.35 SET_TRNS_TPE Built-In Procedure .....	504
A.19.36 SET_TSK_ATTR Built-In Procedure .....	504
A.19.37 SET_TSK_NAME Built-In Procedure .....	505
A.19.38 SET_VAR Built-In Procedure .....	506
A.19.39 %SHADOWVARS Translator Directive .....	508
A.19.40 SHORT Data Type .....	508
A.19.41 SIGNAL EVENT Action .....	509
A.19.42 SIGNAL EVENT Statement .....	509
A.19.43 SIGNAL SEMAPHORE Action .....	509
A.19.44 SIN Built-In Function .....	510
A.19.45 SQRT Built-In Function .....	510
A.19.46 %STACKSIZE Translator Directive .....	511
A.19.47 STD_PTH_NODE Data Type .....	511
A.19.48 STOP Action .....	511
A.19.49 STOP Statement .....	512
A.19.50 STRING Data Type .....	512
A.19.51 STR_LEN Built-In Function .....	513
A.19.52 STRUCTURE Data Type .....	514
A.19.53 SUB_STR Built-In Function .....	514

A.20 - T - KAREL LANGUAGE DESCRIPTION .....	515
A.20.1 TAN Built-In Function .....	515
A.20.2 %TIMESLICE Translator Directive .....	515
A.20.3 %TPMOTION Translator Directive .....	516
A.20.4 TRANSLATE Built-In Procedure .....	516
A.20.5 TRUNC Built-In Function .....	517
A.21 - U - KAREL LANGUAGE DESCRIPTION .....	518
A.21.1 UNHOLD Action .....	518
A.21.2 UNHOLD Statement .....	518
A.21.3 UNINIT Built-In Function .....	519
A.21.4 %UNINITVARS Translator Directive .....	519
A.21.5 UNLOCK_GROUP Built-In Procedure .....	519
A.21.6 UNPAUSE Action .....	521
A.21.7 UNPOS Built-In Procedure .....	521
A.21.8 USING ... ENDUSING Statement .....	522
A.22 - V - KAREL LANGUAGE DESCRIPTION .....	522
A.22.1 V_ACQ_VAMAP iRVision Built-In Procedure.....	522
A.22.2 V_ADJ_2D iRVision Built-In Procedure.....	523
A.22.3 V_CAM_CALIB iRVision Built-In Procedure .....	524
A.22.4 V_CAM_CHECK iRVision Built-In Procedure.....	525
A.22.5 V_CLR_VAMAP iRVision Built-In Procedure.....	526
A.22.6 V_CSAPI_GETVALUE iRVision Built-In Procedure.....	526
A.22.7 V_CSAPI_NUMSET iRVision Built-In Procedure.....	527
A.22.8 V_CSAPI_RESETDATA iRVision Built-In Procedure.....	527
A.22.9 V_CSAPI_SAVEDATA iRVision Built-In Procedure.....	528
A.22.10 V_CSAPI_SETVALUE iRVision Built-In Procedure.....	528
A.22.11 V_CSAPI_TESTRUN iRVision Built-In Procedure.....	529
A.22.12 V_DISPLAY4D iRVision Built-In Procedure.....	530
A.22.13 V_FIND_VIEW iRVision Built-In Procedure.....	530
A.22.14 V_FIND_VLINE iRVision Built-In Procedure.....	531
A.22.15 V_GET_FOUND iRVision Built-In Procedure.....	534
A.22.16 V_GET_OFFSET iRVision Built-In Procedure .....	535
A.22.17 V_GET_PASSFL iRVision Built-In Procedure .....	536
A.22.18 V_GET_READ iRVision Built-In Procedure .....	537
A.22.19 V_GET_VPARAM iRVision Built-In Procedure .....	538
A.22.20 V_IRCONNECT iRVision Built-In Procedure .....	539
A.22.21 V_LED_OFF iRVision Built-In Procedure .....	540
A.22.22 V_LED_ON iRVision Built-In Procedure .....	540
A.22.23 V_OVERRIDE iRVision Built-In Procedure .....	541
A.22.24 V_RUN_FIND iRVision Built-In Procedure .....	541
A.22.25 V_SAVE_IMREG iRVision Built-In Procedure .....	543
A.22.26 V_SET_REF iRVision Built-In Procedure .....	544
A.22.27 V_SNAP_VIEW iRVision Built-In Procedure .....	545
A.22.28 VAR_INFO Built-In Procedure .....	546
A.22.29 VAR_LIST Built-In Procedure .....	548
A.22.30 VECTOR Data Type .....	550
A.22.31 VOL_SPACE Built-In Procedure .....	550
A.22.32 VREG_FND_POS iRVision Built-in Procedure .....	552
A.22.33 VREG_OFFSET iRVision Built-in Procedure .....	552
A.22.34 VT_ACK_QUEUE iRVision Built-In Procedure .....	553
A.22.35 VT_CLR_QUEUE iRVision Built-In Procedure .....	553
A.22.36 VT_DELETE_PQ iRVision Built-In Procedure .....	554
A.22.37 VT_GET_AREID iRVision Built-In Procedure .....	556

A.22.38 VT_GET_FOUND iRVision Built-In Procedure.....	556
A.22.39 VT_GET_LINID iRVision Built-In Procedure.....	557
A.22.40 VT_GET_PFRT iRVision Built-In Procedure.....	557
A.22.41 VT_GET_QUEUE iRVision Built-In Procedure.....	559
A.22.42 VT_GET_TIME iRVision Built-In Procedure.....	560
A.22.43 VT_GET_TRYID iRVision Built-In Procedure.....	560
A.22.44 VT_PUT_QUE2 iRVision Built-In Procedure.....	561
A.22.45 VT_PUT_QUEUE iRVision Built-In Procedure.....	562
A.22.46 VT_READ_PQ iRVision Built-In Procedure.....	562
A.22.47 VT_SET_FLAG iRVision Built-In Procedure.....	567
A.22.48 VT_SET_LDBAL iRVision Built-In Procedure.....	568
A.22.49 VT_WRITE_PQ iRVision Built-In Procedure.....	569
<b>A.23 - W - KAREL LANGUAGE DESCRIPTION .....</b>	<b>571</b>
A.23.1 WAIT FOR Statement .....	571
A.23.2 WHEN Clause .....	571
A.23.3 WHILE...ENDWHILE Statement .....	572
A.23.4 WITH Clause .....	572
A.23.5 WRITE Statement .....	573
A.23.6 WRITE_DICT Built-In Procedure .....	573
A.23.7 WRITE_DICT_V Built-In Procedure .....	574
<b>A.24 - X - KAREL LANGUAGE DESCRIPTION .....</b>	<b>575</b>
A.24.1 XML_ADDTAG Built-In Procedure .....	575
A.24.2 XML_GETDATA Built-In Procedure .....	576
A.24.3 XML_REMTAG Built-In Procedure .....	577
A.24.4 XML_SCAN Built-In Procedure .....	577
A.24.5 XML_SETVAR Built-In Procedure .....	578
A.24.6 XYZWPR Data Type .....	579
A.24.7 XYZWPREXT Data Type .....	579
<b>A.25 - Y - KAREL LANGUAGE DESCRIPTION .....</b>	<b>580</b>
<b>A.26 - Z - KAREL LANGUAGE DESCRIPTION .....</b>	<b>580</b>
<b>B KAREL EXAMPLE PROGRAMS .....</b>	<b>581</b>
B.1 SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING .....	585
B.2 COPYING PATH VARIABLES .....	594
B.3 SAVING DATA TO THE DEFAULT DEVICE .....	602
B.4 STANDARD ROUTINES .....	604
B.5 USING REGISTER BUILT-INS .....	606
B.6 POSITION DATA SET AND CONDITION HANDLERS PROGRAM.....	610
B.7 LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS .....	614
B.8 USING THE FILE AND DEVICE BUILT-INS .....	619
B.9 USING DYNAMIC DISPLAY BUILT-INS .....	622
B.10 MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES .....	630
B.11 DISPLAYING A LIST FROM A DICTIONARY FILE .....	632
B.11.1 Dictionary Files .....	640
B.12 USING THE DISCTRL_ALPHA BUILT-IN .....	641
B.12.1 Dictionary Files .....	644
B.13 APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM .....	644
B.14 iRVISION CUSTOM SCREEN USING V_CSAPI BUILT-INS.....	651
<b>C KCL COMMAND ALPHABETICAL DESCRIPTION .....</b>	<b>655</b>
C.1 ABORT command .....	655
C.2 APPEND FILE command .....	656

C.3 APPEND NODE command .....	656
C.4 CHDIR command .....	657
C.5 CLEAR ALL command .....	657
C.6 CLEAR BREAK CONDITION command .....	658
C.7 CLEAR BREAK PROGRAM command .....	658
C.8 CLEAR DICT command .....	658
C.9 CLEAR PROGRAM command .....	659
C.10 CLEAR VARS command .....	659
C.11 COMPRESS DICT command .....	660
C.12 COMPRESS FORM command .....	660
C.13 CONTINUE command .....	660
C.14 COPY FILE command .....	661
C.15 CREATE VARIABLE command .....	661
C.16 DELETE FILE command .....	662
C.17 DELETE NODE command .....	663
C.18 DELETE VARIABLE command .....	663
C.19 DIRECTORY command .....	663
C.20 DISABLE BREAK PROGRAM command .....	664
C.21 DISABLE CONDITION command .....	664
C.22 DISMOUNT command .....	665
C.23 EDIT command .....	665
C.24 ENABLE BREAK PROGRAM .....	665
C.25 ENABLE CONDITION command .....	666
C.26 FORMAT command .....	666
C.27 HELP command .....	666
C.28 HOLD command .....	667
C.29 INSERT NODE command .....	667
C.30 LOAD ALL command .....	668
C.31 LOAD DICT command .....	668
C.32 LOAD FORM command .....	669
C.33 LOAD MASTER command .....	669
C.34 LOAD PROGRAM command .....	669
C.35 LOAD SERVO command .....	670
C.36 LOAD SYSTEM command .....	670
C.37 LOAD TP command .....	671
C.38 LOAD VARS command .....	672
C.39 LOGOUT command .....	673
C.40 MKDIR command .....	673
C.41 MOUNT command .....	673
C.42 MOVE FILE command .....	674
C.43 PAUSE command .....	674
C.44 PURGE command .....	675
C.45 PRINT command .....	675
C.46 RECORD command .....	676
C.47 RENAME FILE command .....	676
C.48 RENAME VARIABLE command .....	676
C.49 RENAME VARS command .....	677
C.50 RESET command .....	678
C.51 RMDIR command .....	678
C.52 RUN command .....	678
C.53 RUNCF command .....	679
C.54 SAVE MASTER command .....	679
C.55 SAVE SERVO command .....	680

C.56 SAVE SYSTEM command .....	680
C.57 SAVE TP command .....	680
C.58 SAVE VARS command .....	681
C.59 SET BREAK CONDITION command .....	682
C.60 SET BREAK PROGRAM command .....	682
C.61 SET CLOCK command .....	683
C.62 SET DEFAULT command .....	683
C.63 SET GROUP command .....	684
C.64 SET LANGUAGE command .....	684
C.65 SET LOCAL VARIABLE command .....	684
C.66 SET PORT command .....	685
C.67 SET TASK command .....	685
C.68 SET TRACE command .....	686
C.69 SET VARIABLE command .....	686
C.70 SET VERIFY command .....	687
C.71 SHOW BREAK command .....	687
C.72 SHOW BUILTINS command .....	688
C.73 SHOW CONDITION command .....	688
C.74 SHOW CLOCK command .....	688
C.75 SHOW CURPOS command .....	689
C.76 SHOW DEFAULT command .....	689
C.77 SHOW DEVICE command .....	689
C.78 SHOW DICTS command .....	689
C.79 SHOW GROUP command .....	690
C.80 SHOW HISTORY command .....	690
C.81 SHOW LANG command .....	690
C.82 SHOW LANGS command .....	690
C.83 SHOW LOCAL VARIABLE command .....	690
C.84 SHOW LOCAL VARS command .....	691
C.85 SHOW MEMORY command .....	692
C.86 SHOW PROGRAM command .....	692
C.87 SHOW PROGRAMS command .....	692
C.88 SHOW SYSTEM command .....	693
C.89 SHOW TASK command .....	693
C.90 SHOW TASKS command .....	693
C.91 SHOW TRACE command .....	694
C.92 SHOW TYPES command .....	694
C.93 SHOW VARIABLE command .....	694
C.94 SHOW VARS command .....	695
C.95 SHOW data_type command .....	695
C.96 SIMULATE command .....	696
C.97 SKIP command .....	696
C.98 STEP OFF command .....	697
C.99 STEP ON command .....	697
C.100 TRANSLATE command .....	698
C.101 TYPE command .....	698
C.102 UNSIMULATE command .....	698
C.103 WAIT command .....	699

<b>D CHARACTER CODES.....</b>	<b>701</b>
-------------------------------	------------

<b>E SYNTAX DIAGRAMS .....</b>	<b>709</b>
--------------------------------	------------

<b>F TRANSLATING BY ROBOGUIDE.....</b>	<b>735</b>
F.1 TRANSLATE KAREL PROGRAMS.....	735
F.1.1 Sample of Newly Adding a KAREL Program.....	735
F.1.2 Sample of Adding an Existing KAREL Program.....	736
F.2 TRANSLATE A DICTIONARY.....	738
F.2.1 Sample of Translating Newly Created Dictionaries.....	738
F.2.2 Sample of Translating Added Existing Dictionaries.....	739
F.2.3 Dictionary Name and Languages.....	741
F.3 LOAD ON THE ROBOT CONTROLLER.....	741

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# 1 KAREL LANGUAGE OVERVIEW

FANUC's KAREL system consists of a robot, a controller and system software. It accomplishes industrial tasks using programs written in the KAREL programming language. KAREL can manipulate data, control and communicate with related equipment and interact with an operator.

The controller with KAREL works with a wide range of robot models to handle a variety of applications. This means common operating, programming, and troubleshooting procedures, as well as fewer spare parts. KAREL systems expand to include a full line of support products such as integral vision, off-line programming, and application-specific software packages.

The KAREL programming language is a practical blend of the logical, English-like features of high-level languages, such as Pascal and PL/1, and the proven factory-floor effectiveness of machine control languages. KAREL incorporates structures and conventions common to high-level languages as well as features developed especially for robotics applications. These KAREL features include:

- Simple and structured data types
- Arithmetic, relational, and Boolean operators
- Control structures for loops and selections
- Condition handlers
- Procedure and function routines
- Input and output operations
- Multi-programming support

This chapter summarizes the KAREL programming language, and describes the KAREL system software and the controller.

## NOTE

For R-30iB Plus, R-30iB, and R-30iB Mate controllers, the KAREL option must be installed on the robot controller in order to load KAREL programs.

## 1.1 KAREL PROGRAMMING LANGUAGE

A KAREL program is made up of declarations and executable statements stored in a source code file. The variable data values associated with a program are stored in a variable file.

KAREL programs are created and edited using ROBOGUIDE, or another editor such as WordPad.

The KAREL language translator turns the source code into an internal format called p-code and generates a p-code file. The translator is provided with ROBOGUIDE. After being translated, the resulting p-code program can be loaded onto the controller using the *KAREL Command Language (KCL)* or the **FILE** menu. Refer to [Appendix F, TRANSLATING BY ROBOGUIDE](#) for more information about translating a KAREL program using ROBOGUIDE.

During loading, the system will create any required variables that are not in RAM and set them uninitialized. When you run the program, the KAREL interpreter executes the loaded p-code instructions.

A KAREL program is composed of the program logic and the program data. Program logic defines a sequence of steps to be taken to perform a specific task. Program data is the task-related information that the program logic uses. In KAREL the program logic is separate from the program data.

Program logic is defined by KAREL executable statements between the BEGIN and the END statements in a KAREL program. Program data includes variables that are identified in the VAR declaration section of a KAREL program by name, data type and storage area in RAM.

Values for program data can be taught using the teach pendant to jog the robot, computed by the program, read from data files, set from within the CRT/KB or teach pendant menu structure, or accepted as input to the program during execution. The data values can change from one execution to the next, but the same program logic is used to manipulate the data.

Program logic and program data are separate in a KAREL program for the following reasons:

- To allow data to be referenced from several places in the same program
- To allow more than one program to reference or share the same data
- To allow a program to use alternative data
- To facilitate the building of data files by an off-line computer-aided design (CAD) system

The executable section of the program contains the data manipulation statements, I/O statements, and routine calls.

The program development cycle is described briefly in the following list. [Section 1.1.1, Creating a Program](#) - [Section 1.1.5, Execution History](#) that follow provide details on each phase.

- Create a program source code file
- Translate the program file.
- Load the program logic and data.
- Execute the program.
- Maintain the execution history of the program.

A log or history of programs that have been executed is maintained by the controller and can be viewed.

 **CAUTION**

1. All programs in this manual are samples. Before installing and executing them, it is your responsibility to confirm them.
2. Note that in some cases, executing a KAREL program affects the performance of the controller.
3. Be careful not to delete needed programs, variable, files, data, and so forth.
4. ROBOGUIDE is useful for translating and debugging a KAREL program. However, be sure to confirm the program on a real robot.

## 1.1.1 Creating a Program

You can create a KAREL program using the editor provided with ROBOGUIDE, or any text editor such as WordPad. The resulting file is called the *source file* or *source code*.

## 1.1.2 Translating a Program

KAREL source files must be translated into internal code, called *p-code*, before they are executed. The KAREL language translator performs this function and also checks for errors in the source code.

The KAREL language translator starts at the first line of the source code and continues until it encounters an error or translates the program successfully. If an error is encountered, the translator tries to continue checking the program, but no p-code will be generated.

You can invoke the translator from within a ROBOGUIDE workcell or using the KTRANS command utility, and the source code you were editing will be translated. After a successful translation, the translator displays a successful translation message and creates a p-code file. The p-code file will use the source code file name and a .pc file type. This file contains an internal representation of the source code and information the system needs to link the program to variable data and routines.

If you invoke the translator from within a ROBOGUIDE workcell and no errors are detected, the resulting p-code will automatically be loaded on the virtual controller. Refer to [Appendix F, TRANSLATING BY ROBOGUIDE](#) for more information about translating a KAREL program using ROBOGUIDE.

If the translator detects any errors, it displays the error messages and the source lines that were being translated. After you have corrected the errors, you can translate the program again.

The translated file cannot contain more than 32766 bytes of p-code. For R-30iB Plus, it has been expanded to 65534 bytes. If you exceed that amount, the translator will error with Code size exceeded maximum. Break program into separately translated units. You need to move your routines into a separate program file. If you have no routines, you need to break up your program into routines that you can move.

### 1.1.3 Loading Program Logic and Data

---

**NOTE**

For R-30iB Plus, R-30iB, and R-30iB Mate controllers, the KAREL option must be installed on the robot controller in order to load KAREL programs.

The p-code for a program is loaded onto a controller where it can be executed. When a program is loaded, a variable data table, containing all the static variables in the program, is created in RAM. The variable data table contains the program identifier, all of the variable identifiers, and the name of the storage area in RAM where the variables are located.

Loading a program also establishes the links between statements and variables. Initially, the values in the variable data table will be uninitialized. If a variable file (.VR) is loaded successfully, the values of any variables will be stored in the variable data storage area (CMOS, DRAM, SHADOW).

Multiple programs are often used to break a large application or problem into smaller pieces that can be developed and tested separately. The KAREL system permits loading of multiple programs. Each program that is loaded has its own p-code structure.

Variable data can be shared among multiple programs. In this case, the KAREL language FROM clause must be specified in the VAR declaration so that the system can perform the link when the program is loaded. This saves the storage required to include multiple copies of the data.

The following limits apply to the number and size of KAREL programs that can be loaded:

- Number of programs is limited to 2704 or available RAM.
- Number of variables per program is limited to 2704 or available RAM.

## 1.1.4 Executing a Program

After you have selected a program from the program list and the p-code and variable files are loaded into RAM, test and debug the program to make sure that it operates as intended.

Program execution begins at the first executable line. A stack of 300 words is allocated unless you specify a stack size. The stack is allocated from available user RAM. Stack usage is described in [Section 5.1.6, Stack Usage](#).

## 1.1.5 Execution History

Each time a program is executed, a log of the nested routines and the line numbers that have been executed can be displayed from KCL with the `SHOW HISTORY` command.

This is useful when a program has paused or been aborted unexpectedly. Execution history displays the sequence of events that led to the disruption.

## 1.1.6 Program Structure

A KAREL program is composed of declaration and executable sections made up of KAREL language statements, as shown in [Figure 1.1.6](#).

```
PROGRAM prog_name
    Translator Directives
    CONST, TYPE, and/or VAR Declarations
    ROUTINE Declarations
BEGIN
    Executable Statements
END prog_name
ROUTINE Declarations
```

**Figure 1.1.6 Structure of a Program**

In [Figure 1.1.6](#), the words shown in uppercase letters are KAREL reserved words, which have dedicated meanings. `PROGRAM`, `CONST`, `TYPE`, `VAR`, and `ROUTINE` indicate declaration sections of the program. `BEGIN` and `END` mark the executable section. Reserved words are described in [Section 2.1.3, Reserved Words](#).

The `PROGRAM` statement, which identifies the program, must be the first statement in any KAREL program. The `PROGRAM` statement consists of the reserved word `PROGRAM` and an identifier of your choice (`prog_name` in [Figure 1.1.6](#)). Identifiers are described in [Section 2.1.4, User-Defined Identifiers](#).

**NOTE**

Your program must reside in a file. The file can, but does not have to, have the same name as the program. This distinction is important because you invoke the translator and load programs with the name of the file containing your program, but you initiate execution of the program and clear the program with the program name.

For example, if a program named `mover` was contained in a file named `transfer`, you would reference the file by `transfer` to translate it, but would use the program name `mover` to execute the program. If both the program and the file were named `mover`, you could use `mover` to translate the file and also to execute the program.

A task created to execute the program and the task name is the name of the program you initiate. The program can call a routine in another program, but the task name does not change.

The identifier used to name the program cannot be used in the program for any other purpose, such as to identify a variable or constant.

The CONST (constant), TYPE (type), and VAR (variable) declaration sections come after the PROGRAM statement. A program can contain any number of CONST, TYPE, and VAR sections. Each section can also contain any number of individual declaration statements. Also, multiple CONST, TYPE, and VAR sections can appear in any order. The number of CONST, TYPE, and VAR sections, and declaration statements are limited only by the amount of memory available.

ROUTINE declarations can follow the CONST, TYPE, and VAR sections. Each routine begins with the reserved word ROUTINE and is similar in syntax to a program. ROUTINE declarations can also follow the executable section of the main program after the END statement.

The executable section must be marked by BEGIN at the beginning and END, followed by the program identifier (`prog_name` in Figure 1.1.6), at the end. The same program identifier must be used in the END statement as in the PROGRAM statement. The executable section can contain any number of executable statements, limited only by the amount of memory available.

**See Also:** [Chapter 2, LANGUAGE ELEMENTS](#) , [Chapter 3, USE OF OPERATORS](#) , and [Chapter 5, ROUTINES](#)

## 1.2 SYSTEM SOFTWARE

Robot systems include a robot and controller electronics. Hardware interfaces and system software support programming, daily operation, maintenance, and troubleshooting.

This section provides an overview of the supported system software and robot models.

Hardware topics are covered in greater detail in the *Maintenance Manual* specific for your robot and controller model.

### 1.2.1 Software Components

System software is the FANUC-supplied software that is executed by the controller CPU, which allows you to operate the robot system. You use the system software to run programs, as well as to perform daily operations, maintenance, and troubleshooting.

The components of the system software include:

- Motion Control - Movement of the tool center point (TCP) from an initial position to a desired destination position
- File System - Storage of data on the RAM disk or peripheral storage devices
- System Variables - Permanently defined variables declared as part of the KAREL system software
- CRT/KB or Teach Pendant Screens - Screens that facilitate operation of the KAREL system
- KCL - KAREL Command Language
- KAREL Interpreter - Executes KAREL programs

**See Also:** Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN) for detailed operation procedures using the CRT/KB and teach pendant screens.

## 1.2.2 Supported Robots

The robot, using the appropriate tooling, performs application tasks directed by the system software and controller. FANUC robot systems support a variety of robots, each designed for a specific type of application.

For a current list of supported robot models, consult your FANUC technical representative.

For more information on your robot see your *Maintenance Manual* for your specific robot type.

## 1.3 CONTROLLER

The controller contains the electronic circuitry and memory required to operate the system. The electronic circuitry, supported by the system software, directs the operation and motion of the robot and allows communication with peripheral devices.

Controller electronics includes a central processing unit (CPU), several types of memory, an *input/output (I/O)* system, and user interface devices. A cabinet houses the controller electronics and the ports to which remote user interface devices and other peripheral devices are connected.

## 1.3.1 Memory

There are three kinds of controller memory:

- Dynamic Random Access Memory (DRAM)
- A limited amount of battery-backed static/random access memory (SRAM)
- Flash Programmable Read Only Memory (FROM)
- SHADOW

In addition, the controller is capable of storing information externally.

### DRAM

DRAM memory is volatile. Memory contents do not retain their stored values when power is removed. DRAM memory is also referred to as temporary memory (TEMP). The system software is executed in DRAM memory. KAREL programs and most KAREL variables are loaded into DRAM and executed from here also.

**NOTE**

Even though DRAM variables are in volatile memory, you can control their value at startup. Any time that a program .VR or .PC file is loaded, the values in DRAM for that program are set to the value in the .VR file. This means that there is not a requirement to re-load the .VR file itself at every startup to set initial values. If the value of that variable changes during normal operation it will revert to the value it was set to the last time the .VR or .PC file was loaded.

If you want the DRAM variables to be uninitialized at startup you can use the IN UNINIT\_DRAM clause on any variable you want to insure is uninitialized at startup. You can use the %UNINITDRAM directive to specify that all the variables in a program are to be uninitialized at startup.

If you have SHADOW variables and DRAM variables in the same KAREL program, there is a possibility that the power up settings of the DRAM variables could change without loading a .PC/VR file. In this case the programmer must pay particular attention to the reliance of the KAREL software on a particular setting of a DRAM variable at startup. Specifically, the DRAM startup values will always retain the values that they had at the end of controlled start. If SHADOW memory is full, the DRAM startup values could be set during normal system operation.

**SRAM**

SRAM memory is nonvolatile. Memory contents retain their stored values when power is removed. SRAM memory is also referred to as CMOS or as permanent memory (PERM).

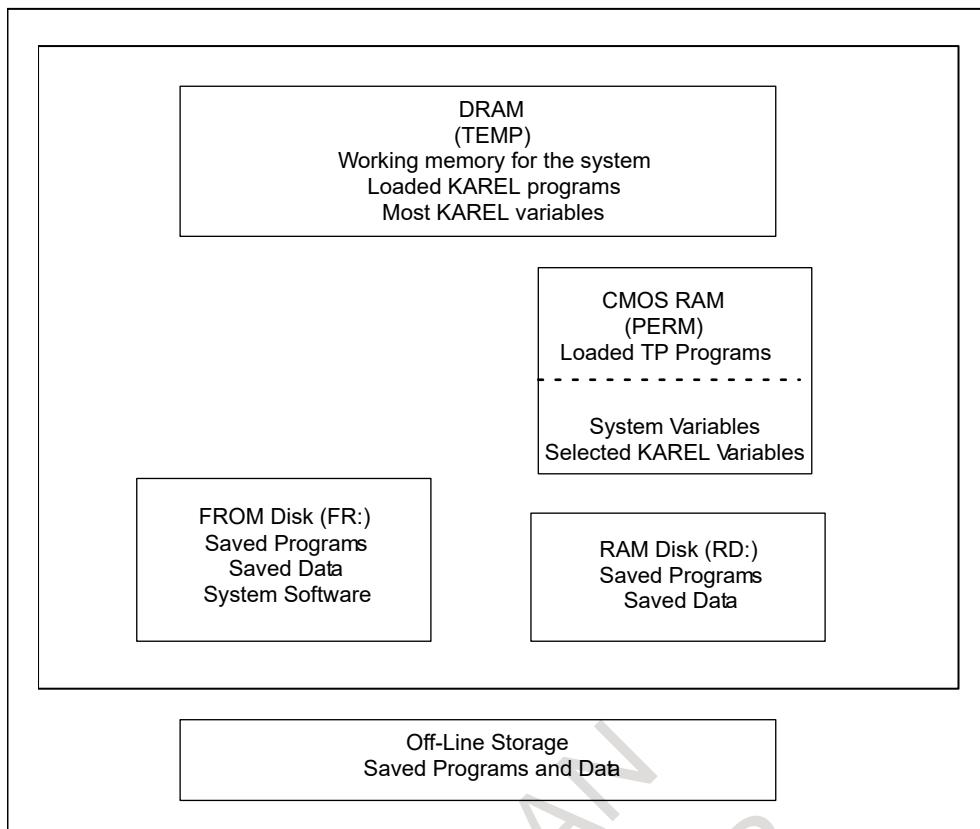
The TPP memory pool (used for teach pendant programs) is allocated from PERM. KAREL programs can designate variables to be stored in CMOS. A portion of SRAM memory can be defined as a user storage device called RAM Disk (RD:).

**Flash Memory (FROM)**

FROM memory is nonvolatile. Memory contents retain their stored values when power is removed. FROM is used for permanent storage of the system software. FROM is also available for user storage as the FROM device (FR:).

**SHADOW**

Shadow memory provides the same capabilities as SRAM. Any values set in shadow are non-volatile and will maintain their state through power cycle. Shadow memory is intended for data which tends to be static. Storing dynamic variables in shadow memory, such as FOR loop indexes or other rapidly changing data, is not efficient.



**Figure 1.3.1 Controller Memory**

### External Storage

You can back up and store files on external devices. You can use the following devices:

- Memory card
- Ethernet via FTP
- USB Memory Stick

## 1.3.2 Input/Output System

The controller can support a modular I/O structure, allowing you to add I/O boards as required by your application. Both digital and analog input and output modules are supported. In addition, you can add optional process I/O boards for additional I/O. The type and number of I/O signals you have depends on the requirements of your application.

See Also: [Chapter 16, INPUT/OUTPUT SYSTEM](#) for more information

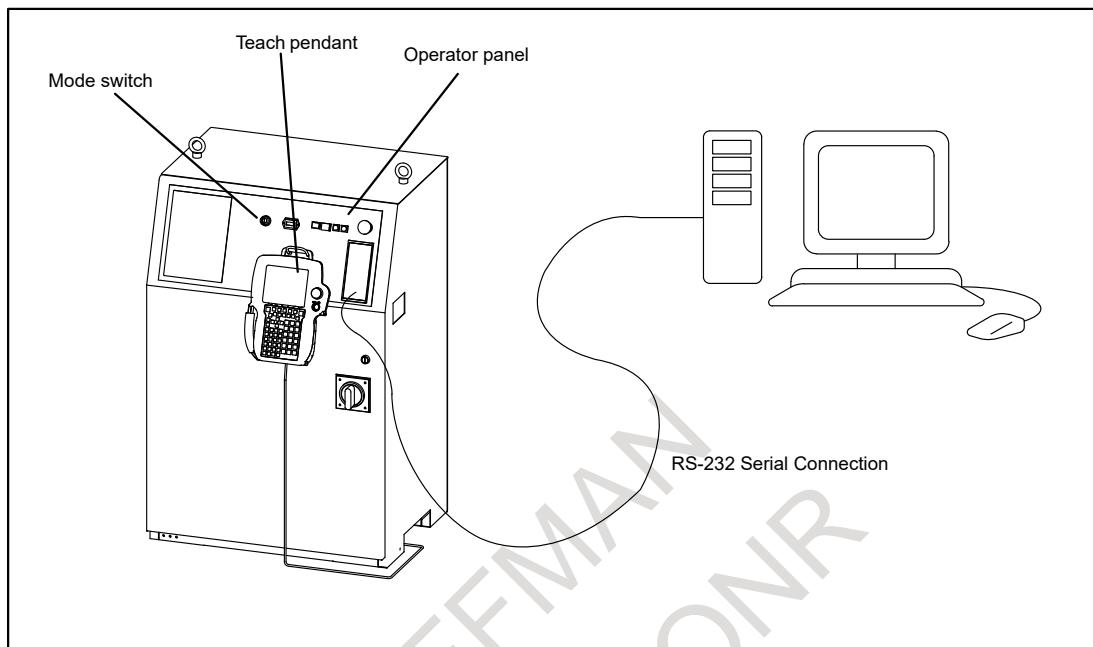
## 1.3.3 User Interface Devices

The user interface devices enable you to program and operate the KAREL system. The common user interface devices supported by KAREL include the operator panel, the teach pendant or the CRT/KB.

**Figure 1.3.3** illustrates these user interface devices. The operator panel and teach pendant have the same basic functions for all models; however, different configurations are also available.

The operator panel, located on the front of the controller cabinet, provides buttons for performing daily operations such as powering up, running a program, and powering down. Lights on the operator panel indicate operating conditions such as when the power is on and when the robot is in cycle.

The system also supports I/O signals for a *user operator panel (UOP)*, which is a user-supplied device such as a custom control panel, a programmable controller, or a host computer. Refer to [Chapter 16, INPUT/OUTPUT SYSTEM](#).



**Figure 1.3.3 Controller**

The CRT/KB is a software option on the controller that allows an external terminal such as a PC running TelNet to display a Menu System that looks similar to the one seen on the teach pendant.

The teach pendant consists of an LCD display, menu-driven function keys, keypad keys, and status LEDs. It is connected to the controller cabinet via a cable, allowing you to perform operations away from the controller.

Internally, the teach pendant connects to the controller's Main CPU board. It is used to jog the robot, teach program data, test and debug programs, and adjust variables. It can also be used to monitor and control I/O, to control end-of-arm tooling, and to display information such as the current position of the robot or the status of an application program.

Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)* provides descriptions of each of the user interface devices, as well as procedures for operating each device.

## 1.3.4 Frames

The value of system variables `$GROUP[group number].$UFRAME` and `$GROUP[group number].$UTOOL` are used as USER and TOOL frames in the treatment associated with position data of KAREL programs. These values must be set to some adequate value by the KAREL program before invoking the function that treats the position data.

The currently selected USER or TOOL frame is used in the motion instruction of a TP program. When using its USER or TOOL frame in a KAREL program, set the value of `$MNUTOOL[group number]`, `$MNUTOOLNUM[group number]` and `$MNUFRAME[group number]`, `$MNUFRAMENUM[group number]` to `$GROUP[].$UTOOL` and `$GROUP[].$UFRAME`.

The position data of a KAREL program does not use the USER or TOOL frame number at the time of teaching; this is different from a TP program. Therefore the position data of the KAREL program is on the USER or TOOL frame value of `$GROUP[group number].$UFRAME` and `$GROUP[group number].$UTOOL` at the time of using it. It is allowed to use `$MOR_GPR[1].$NILPOS` if it is set to the value of WORLD or the mechanical interface coordinate.

EFFMAN  
QUIRIONR

## 2 LANGUAGE ELEMENTS

---

The KAREL language provides the elements necessary for programming effective robotics applications. This chapter lists and describes each of the components of the KAREL language, the available translator directives and the available data types.

### 2.1 LANGUAGE COMPONENTS

---

This section describes the following basic components of the KAREL language:

- Character set
- Operators
- Reserved words
- User-defined Identifiers
- Labels
- Predefined Identifiers
- System Variables
- Comments

#### 2.1.1 Character Set

---

The ASCII character set is available in the KAREL language. [Table 2.1.1 \(a\)](#) lists the elements in the ASCII character set. Three character sets are available in the KAREL language:

- ASCII Character Set
- Multinational Character Set
- Graphics Character Set (not available in R-30iB)

All of the characters recognized by the KAREL language are listed in [Table 2.1.1 \(a\)](#), [Table 2.1.1 \(b\)](#), and [Table 2.1.1 \(c\)](#). The default character set is ASCII. The multinational and graphics character sets are permitted only in literals, data, and comments.

**See Also:** [Section A.3.9, CHR Built-In Function](#)

**Table 2.1.1 (a) ASCII Character Set**

Letters	a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits	0 1 2 3 4 5 6 7 8 9
Symbols	@ < > = / * + - _ , ; : . # \$ ' [ ] ( ) & % { }
Special Characters	blank or space form feed (treated as new line) tab (treated as a blank space)

The following rules are applicable for the ASCII character set:

- Blanks or spaces are:

- Required to separate reserved words and identifiers. For example, the statement PROGRAM prog\_name must include a blank between PROGRAM and prog\_name.
- Allowed but are not required within expressions between symbolic operators and their operands. For example, the statement a = b is equivalent to a=b.
- Used to indent lines in a program.
- Carriage return or a semi-colon (;) separate statements. Carriage returns can also appear in other places.
- A carriage return or a semi-colon is required after the BEGIN statement.
- A line is truncated after 252 characters. It can be continued on the next line by using the concatenation character &.

**Table 2.1.1 (b) Multinational Character Set**

<b>Symbols</b>	í	¢	£	¥	§	¤
	©	a	«	○	±	²
	³	µ	¶	•	¹	º
	»	¼	½	¿		
<b>Special Characters</b>	À	Á	Â	Ã	Ä	Å
	Æ	Ç	È	É	Ê	Ë
	Ì	Í	Î	Ï	Ñ	Ò
	Ó	Ô	Õ	Ö	Œ	Ø
	Ù	Ú	Û	Ü	Ý	Þ
	à	á	â	ã	ä	å
	æ	ç	è	é	ê	ë
	ì	í	î	ï	ñ	ò
	ò	ó	ô	õ	ö	œ
	ø	ù	ú	û	ü	ÿ

**Table 2.1.1 (c) Graphics 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
<b>Digits</b>	0	1	2	3	4	5	6	7	8	9																
<b>Symbols</b>	@	<	>	=	/	*	+	-	,	:	.	#	\$	'	\	[	]	(	)	&	%	!	"	^		
<b>Special Characters</b>	♦	■	H	F	F	C	R	L	F	O	±	N	L	V	T	J	¶	¶	¶	¶	¶	¶	¶	¶	¶	
	+	-	-	-	-	-	-	-	-	†	†	†	†	†	†	†	†	†	†	†	†	†	†	†	†	
	≠	£	.																							

See Also: [Appendix D, CHARACTER CODES](#) for a listing of the character codes for each character set.

## 2.1.2 Operators

KAREL provides operators for standard arithmetic operations, relational operations, and Boolean (logical) operations. KAREL also includes special operators that can be used with positional and VECTOR data types as operands.

[Table 2.1.2 \(a\)](#) lists all of the operators available for use with KAREL.

**Table 2.1.2 (a) KAREL Operators**

Arithmetic	+	-	*	/	DIV	MOD
Relational	<	< =	=	< >	> =	>
Boolean	AND	OR	NOT			
Special	> = <	:	#	@		

The precedence rules for these operators are as follows:

- Expressions within parentheses are evaluated first.
- Within a given level of parentheses, operations are performed starting with those of highest precedence and proceeding to those of lowest precedence.
- Within the same level of parentheses and operator precedence, operations are performed from left to right.

[Table 2.1.2 \(b\)](#) lists the precedence levels for the KAREL operators.

**Table 2.1.2 (b) KAREL Operator Precedence**

OPERATOR	PRECEDENCE LEVEL
NOT	High
:, @, #	↓
*, /, AND, DIV, MOD	↓
Unary + and -, OR, +, -	↓
<, >, =, < >, < =, > =, > <	Low

See Also: [Chapter 3, USE OF OPERATORS](#), for descriptions of functions operators perform

## 2.1.3 Reserved Words

Reserved words have a dedicated meaning in KAREL. They can be used only in their prescribed contexts. All KAREL reserved words are listed in [Table 2.1.3](#).

**Table 2.1.3 Reserved Word List**

ABORT	CONST	GET_VAR	NOPAUSE	STOP
ABOUT	CONTINUE	GO	NOT	STRING
ABS	COORDINATED	GOTO	NOWAIT	STRUCTURE
AFTER	CR	GROUP	OF	THEN
ALONG	DELAY	GROUP_ASSOC	OPEN	TIME
ALSO	DISABLE	HAND	OR	TIMER

AND	DISCONNECT	HOLD	PATH	TO
ARRAY	DIV	IF	PATHHEADER	TPENABLE
ARRAY_LEN	DO	IN	PAUSE	TYPE
AT	DOWNT0	INDEPENDENT	POSITION	UNHOLD
ATTACH	DRAM	INTEGER	POWERUP	UNINIT
AWAY	ELSE	JOINTPOS	PROGRAM	UNPAUSE
AXIS	ENABLE	JOINTPOS1	PULSE	UNTIL
BEFORE	END	JOINTPOS2	PURGE	USING
BEGIN	ENDCONDITION	JOINTPOS3	READ	VAR
BOOLEAN	ENDFOR	JOINTPOS4	REAL	VECTOR
BY	ENDIF	JOINTPOS5	RELATIVE	VIA
BYNAME	ENDMOVE	JOINTPOS6	RELAX	VIS_PROCESS
BYTE	ENDSELECT	JOINTPOS7	RELEASE	WAIT
CAM_SETUP	ENDSTRUCTURE	JOINTPOS8	REPEAT	WHEN
CANCEL	ENDUSING	JOINTPOS9	RESTORE	WHILE
CASE	ENDWHILE	MOD	RESUME	WITH
CLOSE	ERROR	MODEL	RETURN	WRITE
CMOS	EVAL	MOVE	ROUTINE	XYZWPR
COMMAND	EVENT	NEAR	SELECT	XYZWPREXT
COMMON_ASSOC	END	NOABORT	SEMAPHORE	
CONDITION	FILE	NODE	SET_VAR	
CONFIG	FOR	NODEDATA	SHORT	
CONNECT	FROM	NOMESSAGE	SIGNAL	

See Also: [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#)

## 2.1.4 User-Defined Identifiers

User-defined identifiers represent constants, data types, statement labels, variables, routine names, and program names. Identifiers

- Start with a letter
- Can include letters, digits, and underscores
- Can have a maximum of 12 characters (R-30iB and earlier)
- Can have a maximum of 36 characters (R-30iB Plus)
- Can have only one meaning within a particular scope. Refer to [Section 5.1.4, Scope of Variables](#).
- Cannot be reserved words
- Must be defined before they can be used.

For example, the program excerpt in [Figure 2.1.4](#) shows how to declare program, variable, and constant identifiers.

```
PROGRAM mover --program identifier (mover)
  VAR
    original      : POSITION --variable identifier (original)
  CONST
    no_of_parts = 10 --constant identifier (no_of_parts)
```

**Figure 2.1.4 Declaring Identifiers**

## 2.1.5 Labels

Labels are special identifiers that mark places in the program to which program control can be transferred using the GO TO statement.

- Are immediately followed by two colons ( :: ). Executable statements are permitted on the same line and subsequent lines following the two colons.
- Cannot be used to transfer control into or out of a routine.

In [Figure 2.1.5](#), weld:: denotes the section of the program in which a part is welded. When the statement GOTO weld is executed, program control is transferred to the weld section.

```
weld:: --label
  .
  --additional program statements
  .
  .
GOTO weld
```

**Figure 2.1.5 Using Labels**

## 2.1.6 Predefined Identifiers

Predefined identifiers within the KAREL language have a predefined meaning. These can be constants, types, variables, or built-in routine names. [Table 2.1.6 \(a\)](#) and [Table 2.1.6 \(b\)](#) list the predefined identifiers along with their corresponding values. Either the identifier or the value can be specified in the program statement. For example, \$MOTYPE = 7 is the same as \$MOTYPE = LINEAR. However, the predefined identifier MININT is an exception to this rule. This identifier must always be used in place of its value, -2147483648. The value or number itself can not be used.

**Table 2.1.6 (a) Predefined Identifier and Value Summary**

Predefined Identifier	Type	Value
TRUE	BOOLEAN	ON
FALSE		OFF
ON	BOOLEAN	ON
OFF		OFF

Predefined Identifier	Type	Value
MAXINT	INTEGER	+2147483647
MININT		-2147483648
RSWORLD	Orientation Type: \$ORIENT_TYPE	1 2 3
AESWORLD		
WRISTJOINT		
JOINT	Motion Type: \$MOTYPE	6 7 8
LINEAR (or STRAIGHT)		
CIRCULAR		
FINE	Termination Types: \$TERMTYPE and \$SEGTERMTYPE	1 2 3 4 5
COARSE		
NOSETTLE		
NODECEL		
VARDECCEL		

**Table 2.1.6 (b) Port and File Predefined Identifier Summary**

Predefined Identifier	Type
DIN (Digital input)	Boolean port array
DOUT (Digital output)	
GIN (Group input)	Integer port array
GOUT (Group output)	
AIN (Analog input)	
AOUT (Analog output)	
TPIN (Teach pendant input)	Boolean port array
TPOUT (Teach pendant output)	
RDI (Robot digital input)	
RDO (Robot digital output)	
OPIN (Operator panel input)	
OPOUT (Operator panel output)	
WDI (Weld input)	
WDOUT (Weld output)	
UIN (User operator panel input)	
UOUT (User operator panel output)	
LDI (Laser digital input)	
LDO (Laser digital output)	
FLG (Flag)	
MRK (Marker)	

Predefined Identifier	Type
LAI (Laser analog input) LAO (Laser analog output)	Integer port array
TPDISPLAY (Teach pendant KAREL display)* TPERROR (Teach pendant message line) TPPROMPT (Teach pendant function key line)* TPFUNC (Teach pendant function key line)* TPSTATUS (Teach pendant status line)* INPUT (CRT/KB KAREL keyboard)* OUTPUT (CRT/KB KAREL screen)* CRTERROR (CRT/KB message line) CRTFUNC (CRT function key line)* CRTSTATUS (CRT status line)* CRTPROMPT (CRT prompt line)* VIS_MONITOR (Vision Monitor Screen)	File

\*Input and output occurs on the **USER** menu of the teach pendant or CRT/KB.

## 2.1.7 System Variables

---

System variables are variables that are declared as part of the KAREL system software. They have permanently defined variable names, that begin with a dollar sign (\$). Many are robot specific, meaning their values depend on the type of robot that is attached to the system.

Some system variables are not accessible to KAREL programs. Access rights govern whether or not a KAREL program can read from or write to system variables.

## 2.1.8 Comments

---

Comments are lines of text within a program used to make the program easier for you or another programmer to understand. For example, Figure 2.1.8 contains some comments from Figure 2.2 (a) and Figure 2.2 (b).

```
--This program, called mover, picks up 10 objects
--from an original POSITION and puts them down
--at a destination POSITION.
original : POSITION --POSITION of objects
destination : POSITION --Destination of objects
count   : INTEGER    --Number of objects moved
```

**Figure 2.1.8 Comments From Within a Program**

A comment is marked by a pair of consecutive hyphens (--). On a program line, anything to the right of these hyphens is treated as a comment.

Comments can be inserted on lines by themselves or at the ends of lines containing any program statement. They are ignored by the translator and have absolutely no effect on a running program.

## 2.2 TRANSLATOR DIRECTIVES

---

Translator directives provide a mechanism for directing the translation of a KAREL program. Translator directives are special statements used within a KAREL program to

- Include other files into a program at translation time
- Specify program and task attributes

All directives except %INCLUDE must be after the program statement but before any other statements.

[Table 2.2 lists and briefly describes each translator directive. Refer to Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION for a complete description of each translator directive.](#)

**Table 2.2 Translator Directives**

Directive	Description
%ALPHABETIZE	Specifies that variables will be created in alphabetical order when p-code is loaded.
%CMOSVARS	Specifies the default storage for KAREL variables is CMOS RAM.
%CMOS2SHADOW	Instructs the translator to put all CMOS variables in SHADOW memory.
%COMMENT = 'comment'	Specifies a comment of up to 16 characters. During load time, the comment is stored as a program attribute and can be displayed on the <b>SELECT</b> screen of the teach pendant or CRT/KB.
%CRTDEVICE	Specifies that the CRT/KB user window will be the default in the READ and WRITE statements instead of the TPDISPLAY window.
%DEFGROUP = n	Specifies the default motion group to be used by the translator.
%DELAY	Specifies the amount of time the program will be delayed out of every 250 milliseconds.
%ENVIRONMENT filename	Used by the off-line translator to specify that a particular environment file should be loaded.
%INCLUDE filename	Specifies files to insert into a program at translation time.
%LOCKGROUP =n,n	Specifies the motion group(s) locked by this task.
%NOABORT = option	Specifies a set of conditions which will be prevented from aborting the program.
%NOBUSYLAMP	Specifies that the busy lamp will be OFF during execution.
%NOLOCKGROUP	Specifies that no motion groups will be locked by this task.
%NOPAUSE = option	Specifies a set of conditions which will be prevented from pausing the program.
%NOPAUSESHT	Specifies that the task is not paused if the teach pendant shift key is released.
%PRIORITY = n	Specifies the task priority.
%SHADOWVARS	Specifies that all variables by default are created in SHADOW.
%STACKSIZE = n	Specifies the stack size in long words.

Directive	Description
%TIMESLICE = n	Supports round-robin type time slicing for tasks with the same priority.
%TPMOTION	Specifies that task motion is enabled only when the teach pendant is enabled.
%UNINITVARS	Specifies that all variables are by default uninitialized.

Figure 2.2 (a) illustrates the %INCLUDE directive. Figure 2.2 (b) shows the included file.

```

PROGRAM mover
-- This program, called mover, picks up 10 objects
-- from an original position and puts them down
-- at a destination position.
%INCLUDE mover_decs
-- Uses %INCLUDE directive to include the file
-- called mover_decs containing declarations
BEGIN
    OPEN HAND gripper
    -- Loop to move total number of objects
    FOR count = 1 TO num_of_parts DO
        -- Put position in Position Register 1
        SET_POS_REG(1,original,status)
        -- Call TP program to move to Position Register
        move_to_pr
        CLOSE HAND gripper
        SET_POS_REG(1,destination,status)
        move_to_pr
        OPEN HAND gripper
    ENDFOR      -- End of loop
END mover
The TP program move_to_pr is a one line program
to do the move:
1:J PR[1] 100% FINE

```

Figure 2.2 (a) %INCLUDE Directive in a KAREL Program

```

-- Declarations for program mover in file mover_decs
VAR
    original : XYZWPR      --POSITION of objects
    destination : XYZWPR     --Destination of objects
    count : INTEGER          --Number of objects moved
CONST
    gripper = 1      -- Hand number 1
    num_of_parts = 10     -- Number of objects to move

```

Figure 2.2 (b) Include File mover\_decs for a KAREL Program

## 2.3 DATA TYPES

Three forms of data types are provided by KAREL to define data items in a program:

- Simple type data items
  - Can be assigned constants or variables in a KAREL program
  - Can be assigned actual (literal) values in a KAREL program

- Can assume only single values
- Structured type data items
  - Are defined as data items that permit or require more than a single value
  - Are composites of simple data and structured data
- User-defined type data items
  - Are defined in terms of existing data types including other user-defined types
  - Can be defined as structures consisting of several KAREL variable data types
  - Cannot include itself

Table 2.3 lists the simple and structured data types available in KAREL. User-defined data types are described in Section 2.4, [USER-DEFINED DATA TYPES AND STRUCTURES](#).

**Table 2.3 Simple and Structured Data Types**

Simple	Structured	
BOOLEAN	ARRAY OF BYTE	JOINTPOS8
FILE	CAM_SETUP	JOINTPOS9
INTEGER	CONFIG	MODEL
REAL	JOINTPOS	PATH
STRING	JOINTPOS1	POSITION
	JOINTPOS2	QUEUE_TYPE
	JOINTPOS3	ARRAY OF SHORT
	JOINTPOS4	VECTOR
	JOINTPOS5	VIS_PROCESS
	JOINTPOS6	XYZWPR
	JOINTPOS7	XYZWPREXT

See Also: [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for a detailed description of each data type.

## 2.4 USER-DEFINED DATA TYPES AND STRUCTURES

User-defined data types are data types you define in terms of existing data types. User-defined data structures are data structures in which you define a new data type as a structure consisting of several KAREL variable data types, including previously defined user data types.

### 2.4.1 User-Defined Data Types

User-defined data types are data types you define in terms of existing data types. With user-defined data types, you

- Include their declarations in the TYPE sections of a KAREL program.
- Define a KAREL name to represent a new data type, described in terms of other data types.

- Can use predefined data types required for specific applications.

User-defined data types can be defined as structures, consisting of several KAREL variable data types.

The continuation character, &, can be used to continue a declaration on a new line.

[Figure 2.4.1](#) shows an example of user-defined data type usage and continuation character usage.

```

CONST
    n_pages = 20
    n_lines = 40
    std_str_lng = 8
TYPE
    std_string_t = STRING [std_str_lng]
    std_table_t = ARRAY [n_pages]&           --continuation character
        OF ARRAY [n_lines] OF std_string_t
    path_hdr_t FROM main_prog = STRUCTURE   --user defined data type
        ph_uframe: POSITION
        ph_utool: POSITION
    ENDSTRUCTURE
    node_data_t FROM main_prog = STRUCTURE
        gun_on: BOOLEAN
        air_flow: INTEGER
    ENDSTRUCTURE
    std_path_t FROM main_prog =
        PATH PATHDATA = path_hdr_t NODEDATA = node_data_t
VAR
    msg_table_1: std_table_t
    msg_table_2: std_table_t
    temp_string: std_string_t
    seam_1_path: std_path_t

```

**Figure 2.4.1 User-Defined Data Type Example**

## Usage

User-defined type data can be

- Assigned to other variables of the same type
- Passed as a parameter
- Returned as a function

Assignment between variables of different user-defined data types, even if identically declared, is not permitted. In addition, the system provides the ability to load and save variables of user-defined data types, checking consistency during the load with the current declaration of the data type.

## Restrictions

A user-defined data type cannot

- Include itself
- Include any type that includes it, either directly or indirectly
- Be declared within a routine

## 2.4.2 User-Defined Data Structures

A structure is used to store a collection of information that is generally used together. User-defined data structures are data structures in which you define a new data type as a structure consisting of several KAREL variable data types.

When a program containing variables of user-defined data types is loaded, the definitions of these types are checked against a previously created definition. If a previously created definition does not exist, a new one is created.

With user-defined data structures, you

- Define a data type as a structure consisting of a list of component fields, each of which can be a standard data type or another, previously defined, user data type. See [Figure 2.4.2 \(a\)](#).

```
new_type_name = STRUCTURE
    field_name_1: type_name_1
    field_name_2: type_name_2
    ..
ENDSTRUCTURE
```

**Figure 2.4.2 (a) Defining a Data Type as a User-Defined Structure**

- Access elements of a data type defined as a structure in a KAREL program. The continuation character, &, can be used to continue access of the structure elements. See [Figure 2.4.2 \(b\)](#).

```
var_name = new_type_name.field_name_1
new_type_name.field_name_2 = expression
outer_struct_name.inner_struct_name&
    .field_name = expression
```

**Figure 2.4.2 (b) Accessing Elements of a User-Defined Structure in a KAREL Program**

- Access elements of a data type defined as a structure from the CRT/KB and at the teach pendant.
- Define a range of executable statements in which fields of a STRUCTURE type variable can be accessed without repeating the name of the variable. See [Figure 2.4.2 \(c\)](#).

```
USING struct_var, struct_var2 DO
    statements
    ..
ENDUSING
```

**Figure 2.4.2 (c) Defining a Range of Executable Statements**

In the above example, *struct\_var* and *struct\_var2* are the names of structure type variables.

### NOTE

If the same name is both a field name and a variable name, the field name is assumed. If the same field name appears in more than one variable, the right-most variable in the USING statement is used.

### Restrictions

User-defined data structures have the following restrictions:

- The following data types are not valid as part of a data structure:

- STRUCTURE definitions; types that are declared structures are permitted. See [Figure 2.4.2 \(d\)](#).

```
The following is valid:
TYPE
    sub_struct = STRUCTURE
        subs_field_1: INTEGER
        subs_field_2: BOOLEAN
    ENDSTRUCTURE
    big_struct = STRUCTURE
        bigs_field_1: INTEGER
        bigs_field_2: sub_struct
    ENDSTRUCTURE
The following is not valid:
big_struct = STRUCTURE
    bigs_field_1: INTEGER
    bigs_field_2: STRUCTURE
    subs_field_1: INTEGER
    subs_field_2: BOOLEAN
ENDSTRUCTURE
ENDSTRUCTURE
```

**Figure 2.4.2 (d) Valid STRUCTURE Definitions**

- PATH types
- FILE types
- VISION types
- Variable length arrays
- The data structure itself, or any type that includes it, either directly or indirectly
- Any structure not previously defined.
- A variable can not be defined as a structure, but can be defined as a data type previously defined as a structure. See [Figure 2.4.2 \(e\)](#).

```
The following is valid:
TYPE
    struct_t = STRUCTURE
        st_1: BOOLEAN
        st_2: REAL
    ENDSTRUCTURE
VAR
    var_name: struct_t
The following is not valid:
VAR
    var_name: STRUCTURE
        vn_1: BOOLEAN
        vn_2: REAL
    ENDSTRUCTURE
```

**Figure 2.4.2 (e) Defining a Variable as a Type Previously Defined as a Structure**

## 2.5 ARRAYS

You can declare arrays of any data type except PATH.

You can access elements of these arrays in a KAREL program, from the CRT/KB, and from the teach pendant.

In addition, you can define two types of arrays:

- Multi-dimensional arrays
- Variable-sized arrays

## 2.5.1 Multi-Dimensional Arrays

Multi-dimensional arrays are arrays of elements with two or three dimensions. These arrays allow you to identify an element using two or three subscripts.

Multi-dimensional arrays allow you to

- Declare variables as arrays with two or three (but not more) dimensions. See [Figure 2.5.1 \(a\)](#).

```
VAR
  name: ARRAY [size_1] OF ARRAY [size_2] ... , OF element_type
  OR
VAR
  name: ARRAY [size_1, size_2,...] OF element_type
```

**Figure 2.5.1 (a) Declaring Variables as Arrays with Two or Three Dimensions**

- Access elements of these arrays in KAREL statements. See [Figure 2.5.1 \(b\)](#).

```
name [subscript_1, subscript_2,...] = value
value = name [subscript_1, subscript_2,...]
```

**Figure 2.5.1 (b) Accessing Elements of Multi-Dimensional Arrays in KAREL Statements**

- Declare routine parameters as multi-dimensional arrays. See [Figure 2.5.1 \(c\)](#).

```
Routine expects 2-dimensional array of INTEGER.
ROUTINE array_user (array_param:ARRAY [*,*] OF INTEGER)
The following are equivalent:
ROUTINE rtn_name(array_param: ARRAY[*] OF INTEGER)
and
ROUTINE rtn_name(array_param: ARRAY OF INTEGER)
```

**Figure 2.5.1 (c) Declaring Routine Parameters as Multi-Dimensional Arrays**

- Access elements with KCL commands and the teach pendant.
- Show elements in the KAREL variable screen.
- Save and load multi-dimensional arrays to and from variable files.

### Restrictions

The following restrictions apply to multi-dimensional arrays:

- A subarray can be passed as a parameter or assigned to another array by omitting one or more of the right-most subscripts only if it was defined as a separate type. See [Figure 2.5.1 \(d\)](#).

```
TYPE
  array_30 = ARRAY[30] OF INTEGER
  array_20_30 = ARRAY[20] OF array_30
VAR
  array_1: array_30
  array_2: array_20_30
  array_3: ARRAY[10] OF array_20_30
ROUTINE array_user(array_data: ARRAY OF INTEGER
  FROM other-prog
```

```

BEGIN
array_2 = array_3[10]           -- assigns elements array_3[10,1,1]
                                -- through array_3[10,20,30] to
array_2
array_2[2] = array_1            -- assigns elements array_1[1]
through
                                -- array_1 [30] to elements
array_2[2,1]
                                -- through array_2[2,30]
array_user(array_3[5,3])        -- passes elements array_3[5,3,1]
                                -- through array_3[5,3,30] to
array_user

```

**Figure 2.5.1 (d) Using a Subarray**

- The element type cannot be any of the following:
  - ARRAY (but it can be a user-defined type that is an array)
  - PATH

## 2.5.2 Variable-Sized Arrays

Variable-sized arrays are arrays whose actual size is not known, and that differ from one use of the program to another. Variable-sized arrays allow you to write KAREL programs without establishing dimensions of the array variables. In all cases, the dimension of the variable must be established before the .PC file is loaded.

Variable-sized arrays allow you to

- Declare an array size as to-be-determined (\*). See [Figure 2.5.2](#).

```

VAR
one_d_array:  ARRAY[*] OF type
two_d_array:  ARRAY[*,*] OF type

```

**Figure 2.5.2 Indicates that the Size of an Array is To-Be-Determined**

- Determine an array size from that in a variable file or from a KCL CREATE VAR command rather than from the KAREL source code.

The actual size of a variable-sized array will be determined by the actual size of the array if it already exists, the size of the array in a variable file if it is loaded first, or the size specified in a KCL CREATE VAR command executed before the program is loaded. Dimensions explicitly specified in a program must agree with those specified from the .VR file or specified in the KCL CREATE VAR command.

### Restrictions

Variable-sized arrays have the following restrictions:

- The variable must be loaded or created in memory (in a .VR file or using KCL), with a known length, before it can be used.
- When the .PC file is loaded, it uses the established dimension, otherwise it uses 0.
- Variable-sized arrays are only allowed in the VAR section and not the TYPE section of a program.
- Variable-sized arrays are only allowed for static variables.

EFFMAN  
QUIRIONR

## 3 USE OF OPERATORS

---

This chapter describes how operators are used with other language elements to perform operations within a KAREL application program. Expressions and assignments, which are program statements that include operators and operands, are explained first. Next, the kinds of operations that can be performed using each available KAREL operator are discussed.

### 3.1 EXPRESSIONS AND ASSIGNMENTS

---

Expressions are values defined by a series of operands, connected by operators and cause desired computations to be made. For example,  $4+8$  is an expression in which  $4$  and  $8$  are the *operands* and the plus symbol ( $+$ ) is the *operator*.

Assignments are statements that set the value of variables to the result of an evaluated expression.

#### 3.1.1 Rule for Expressions and Assignments

---

The following rules apply to expressions and assignments:

- Each operand of an expression has a data type determined by the nature of the operator.
- Each KAREL operator requires a particular operand type and causes a computation that produces a particular result type.
- Both operands in an expression must be of the same data type. For example, the AND operator requires that both its operands are INTEGER values or that both are BOOLEAN values. The expression  $i \text{ AND } b$ , where  $i$  is an INTEGER and  $b$  is a BOOLEAN, is invalid.
- Five special cases in which the operands can be mixed provide an exception to this rule. These five cases include the following:
  - INTEGER and REAL operands to produce a REAL result
  - INTEGER and REAL operands to produce a BOOLEAN result
  - INTEGER and VECTOR operands to produce a VECTOR
  - REAL and VECTOR operands to produce a VECTOR
  - POSITION and VECTOR operands to produce a VECTOR
- Any positional data type can be substituted for the POSITION data type.

#### 3.1.2 Evaluation of Expressions and Assignments

---

[Table 3.1.2](#) summarizes the data types of the values that result from the evaluation of expressions containing KAREL operators and operands.

**Table 3.1.2 Summary of Operation Result Types**

Operator	+	-	*	/	DIV MOD	< >, >= <=, <, >, =	> <=	AND OR NOT	#	@	:
<b>Types of Operators</b>											
INTEGER	I	I	I	R	I	B	-	I	-	-	-
REAL	R	R	R	R	-	B	-	-	-	-	-
Mixed** INTEGER- REAL	R	R	R	R	-	B	-	-	-	-	-
BOOLEAN	-	-	-	-	-	B	-	B	-	-	-
STRING	S	-	-	-	-	B	-	-	-	-	-
Mixed** INTEGER- VECTOR	-	-	V	V	-	-	-	-	-	-	-
Mixed** REAL- VECTOR	-	-	V	V	-	-	-	-	-	-	-
VECTOR	V	V	-	-	-	B***	-	-	V	R	-
POSITION	-	-	-	-	-	-	B	-	-	-	P
Mixed** POSITION- VECTOR	-	-	-	-	-	-	-	-	-	-	V

\*\*Mixed means one operand of each type

\*\*\*VECTOR values can be compared using = < > only

-Operation not allowed

I INTEGER

R REAL

B BOOLEAN

V VECTOR

P POSITION

## 3.1.3 Variables and Expressions

Assignment statements contain variables and expressions. The variables can be any user-defined variable, a system variable with write access, or an output port array with write access. The expression can be any valid KAREL expression. The following examples are acceptable assignments:

```
$KAREL_ENB = 1 -- assigns an INTEGER value to a system variable
```

```
real_var = real_var + 1 -- assigns a REAL value to a REAL variable
```

The data types of variable and expression must match with three exceptions:

- INTEGER variables can be assigned to REAL variables. In this case, the INTEGER is treated as a REAL number during evaluation of the expression. However, a REAL number cannot be used where an INTEGER value is expected.
- If required, a REAL number can be converted to an INTEGER using the ROUND or TRUNC built-in functions.
- INTEGER, BYTE, and SHORT types can be assigned to each other, although a run-time error will occur if the assigned value is out of range.
- Any positional type can be assigned to any other positional type. A run-time error will result if a JOINTPOS from a group without kinematics is assigned to an XYZWPR.

**See Also:** [Section 3.2.2, Relational Operations](#) , [Section A.18.30, ROUND Built-In Function](#) , [Section A.20.5, TRUNC Built-In Function](#)

## 3.2 OPERATIONS

Operations include the manipulation of variables, constants, and literals to compute values using the available KAREL operators. The following operations are discussed:

- Arithmetic Operations
- Relational Operations
- Boolean Operations
- Special Operations

[Table 3.2](#) lists all of the operators available for use with KAREL.

**Table 3.2 KAREL Operators**

Operation	Operator					
Arithmetic	+	-	*	/	DIV	MOD
Relational	<	<=	=	<>	>=	>
Boolean	AND	OR	NOT			
Special	>= <	:	#	@		

## 3.2.1 Arithmetic Operations

The addition (+), subtraction (-), and multiplication (\*) operators, along with the DIV and MOD operators, can be used to compute values within arithmetic expressions. Refer to [Table 3.2.1 \(a\)](#).

**Table 3.2.1 (a) Arithmetic Operations Using +, -, and \* Operators**

EXPRESSION	RESULT
$3 + 2$	5
$3 - 2$	1
$3 * 2$	6

- The DIV and MOD operators are used to perform INTEGER division. Refer to [Table 3.2.1 \(b\)](#).

**Table 3.2.1 (b) Arithmetic Operations Examples**

EXPRESSION	RESULT
11 DIV 2	5
11 MOD 2	1

- The DIV operator truncates the result of an equation if it is not a whole number.
- The MOD operator returns the remainder of an equation that results from dividing the left-side operand by the right-side operand.
- If the right-side operand of a MOD equation is a negative number, the result is also negative.
- If the divisor in a DIV equation or the right-side operand of a MOD equation is zero, the KAREL program is aborted with the Divide by zero error.
- The INTEGER bitwise operators, AND, OR, and NOT, produce the result of a binary AND, OR, or NOT operation on two INTEGER values. Refer to [Table 3.2.1 \(c\)](#).

**Table 3.2.1 (c) Arithmetic Operations Using Bitwise Operands**

EXPRESSION	BINARY EQUIVALENT	RESULT
5 AND 8	0101 AND 1000	0000 = 0
5 OR 8	0101 OR 1000	1101 = 13
-4 AND 8	1100 AND 1000	1000 = 8
-4 OR 8	1100 OR 1000	1100 = -4
NOT 5	NOT 0101	1010 = -6*
NOT -15	NOT 110001	1110 = 14*

\*Because negative INTEGER values are represented in the two's complement form, NOT  $i$  is not the same as  $-i$ .

- If an INTEGER or REAL equation results in a value exceeding the limit for INTEGER or REAL variables, the program is aborted with an error. If the result is too small to represent, it is set to zero.

[Table 3.2.1 \(d\)](#) lists the precedence levels for the KAREL operators.

**Table 3.2.1 (d) KAREL Operator Precedence**

OPERATOR	PRECEDENCE LEVEL
NOT	High

OPERATOR	PRECEDENCE LEVEL
: , @, #	↓
* , /, AND, DIV, MOD	↓
Unary + and -, OR, +, -	↓
<, >, =, < >, <=, >=, > <	Low

## 3.2.2 Relational Operations

Relational operators ( $<>$ ,  $=$ ,  $>$ ,  $<$ ,  $\leq$ ,  $\geq$ ) produce a BOOLEAN (TRUE/FALSE) result corresponding to whether or not the values of the operands are in the relation specified. In a relational expression, both operands must be of the same simple data type. Two exceptions to this rule exist:

- REAL and INTEGER expressions can be mixed where the INTEGER operand is converted to a REAL number.
- For example, in the expression  $1 > .56$ , the number 1 is converted to 1.0 and the result is TRUE.
- VECTOR operands, which are a structured data type, can be compared in a relational expression but only by using the equality (=) or inequality ( $\neq$ ) operators.

The relational operators function with INTEGER and REAL operands to evaluate standard mathematical equations. Refer to [Table 3.2.2](#).

### NOTE

Performing equality (=) or inequality ( $\neq$ ) tests between REAL values might not yield the results you expect. Because of the way REAL values are stored and manipulated, two values that would appear to be equal might not be exactly equal. This is also true of VECTOR values which are composed of REAL values. Use  $\geq$  or  $\leq$  where appropriate instead of  $=$ .

Relational operators can also have STRING values as operands. STRING values are compared lexically character by character from left to right until one of the following occurs. Refer to [Table 3.2.2](#).

- The character code for a character in one STRING is greater than the character code for the corresponding character in the other STRING. The result in this case is that the first string is greater. For example, the ASCII code for A is 65 and for a is 97. Therefore,  $a > A = \text{TRUE}$ .
- One STRING is exhausted while characters remain in the other STRING. The result is that the first STRING is less than the other STRING.
- Both STRING expressions are exhausted without finding a mismatch. The result is that the STRINGS are equal.

**Table 3.2.2 Relational Operation Examples**

EXPRESSION	RESULT
'A' < 'AA'	TRUE
'A' = 'a'	FALSE
4 > 2	TRUE
17.3 < > 5.6	TRUE
(3 * 4) < > (4 * 3)	FALSE

With BOOLEAN operands, TRUE>FALSE is defined as a true statement. Thus the expression FALSE>=TRUE is a false statement. The statements FALSE>=FALSE and TRUE>=FALSE are also true statements.

### 3.2.3 Boolean Operations

The Boolean operators AND, OR, and NOT, with BOOLEAN operands, can be used to perform standard mathematical evaluations. [Table 3.2.3 \(a\)](#) summarizes the results of evaluating Boolean expressions, and some examples are listed in [Table 3.2.3 \(b\)](#).

**Table 3.2.3 (a) BOOLEAN Operation Summary**

OPERATOR	OPERAND 1	OPERAND 2	RESULT
NOT	TRUE	-	FALSE
	FALSE	-	TRUE
OR	TRUE	TRUE	TRUE
		FALSE	
	FALSE	TRUE	FALSE
		FALSE	
AND	TRUE	TRUE	TRUE
		FALSE	FALSE
	FALSE	TRUE	
		FALSE	

**Table 3.2.3 (b) BOOLEAN Operations Using AND, OR, and NOT Operators**

EXPRESSION	RESULT
DIN[1] AND DIN[2]	TRUE if DIN[1] and DIN[2] are both TRUE; otherwise FALSE
DIN[1] AND NOT DIN[2]	TRUE if DIN[1] is TRUE and DIN[2] is FALSE; otherwise FALSE
(x < y) OR (y > z)	TRUE if x < y or if y > z; otherwise FALSE
(i = 2) OR (i = 753)	TRUE if i = 2 or if i = 753; otherwise FALSE

### 3.2.4 Special Operations

The KAREL language provides special operators to perform functions such as testing the value of approximately equal POSITION variables, relative POSITION variables, VECTOR variables, and STRING variables. This section describes their operations and gives examples of their usage.

The following rules apply to approximately equal operations:

- The relational operator ( $>=<$ ) determines if two POSITION operands are approximately equal and produces a BOOLEAN result. The comparison is similar to the equality ( $=$ ) relation except that the operands compared need not be identical. Extended axis values are not considered.

- Approximately equal operations must be used in conjunction with the system variables, *\$LOCTOL*, *\$ORIENTTOL*, and *\$CHECKCONFIG* to determine how close two positions must be.
- The relational operator ( $\geq\leq$ ) is allowed only in normal program use and cannot be used as a condition in a condition handler statement.

In the following example the relational operator ( $\geq\leq$ ) is used to determine if the current robot position (determined by using the CURPOS built-in procedure) is near the designated perch position:

```
IF perch >=< CURPOS (0,0) THEN
  — Call move to perch program
  move_to_perch
ELSE
  ABORT
ENDIF
```

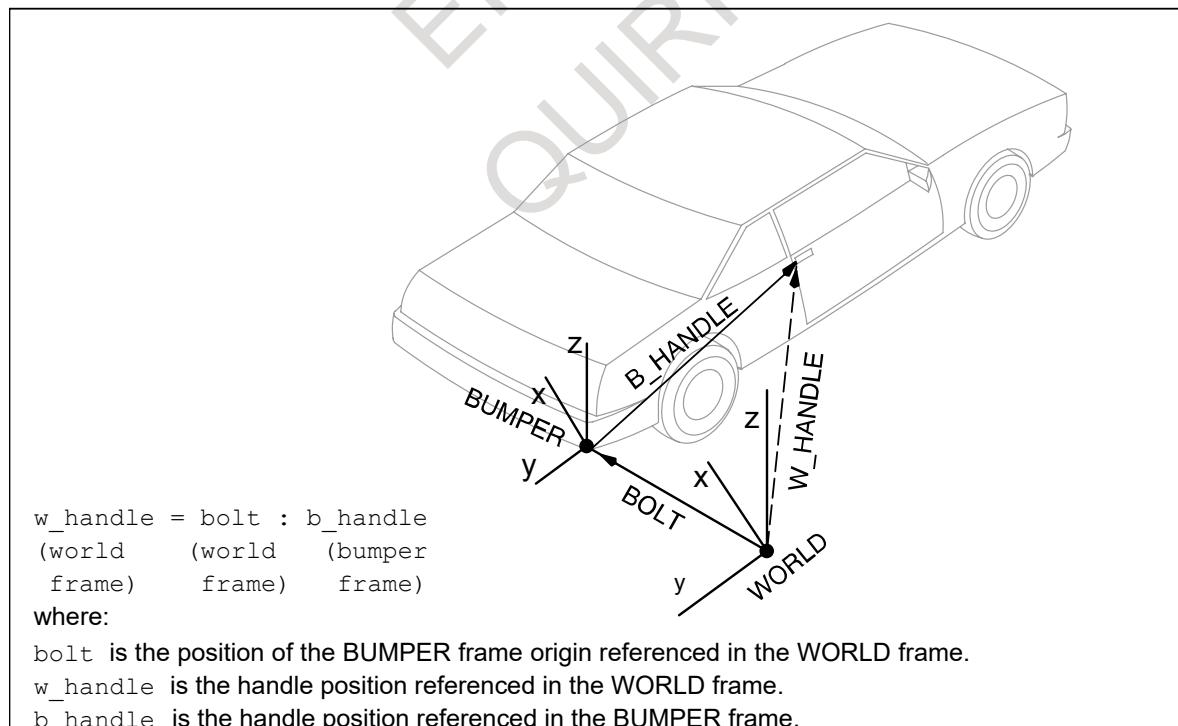
**Figure 3.2.4 (a) Relational Operator**

### Relative Position Operations

To locate a position in space, you must reference it to a specific coordinate frame. In KAREL, reference frames have the POSITION data type. The relative position operator (:) allows you to reference a position or vector with respect to the coordinate frame of another position (that is, the coordinate frame that has the other position as its origin point).

The relative position operator (:) is used to transform a position from one reference frame to another frame.

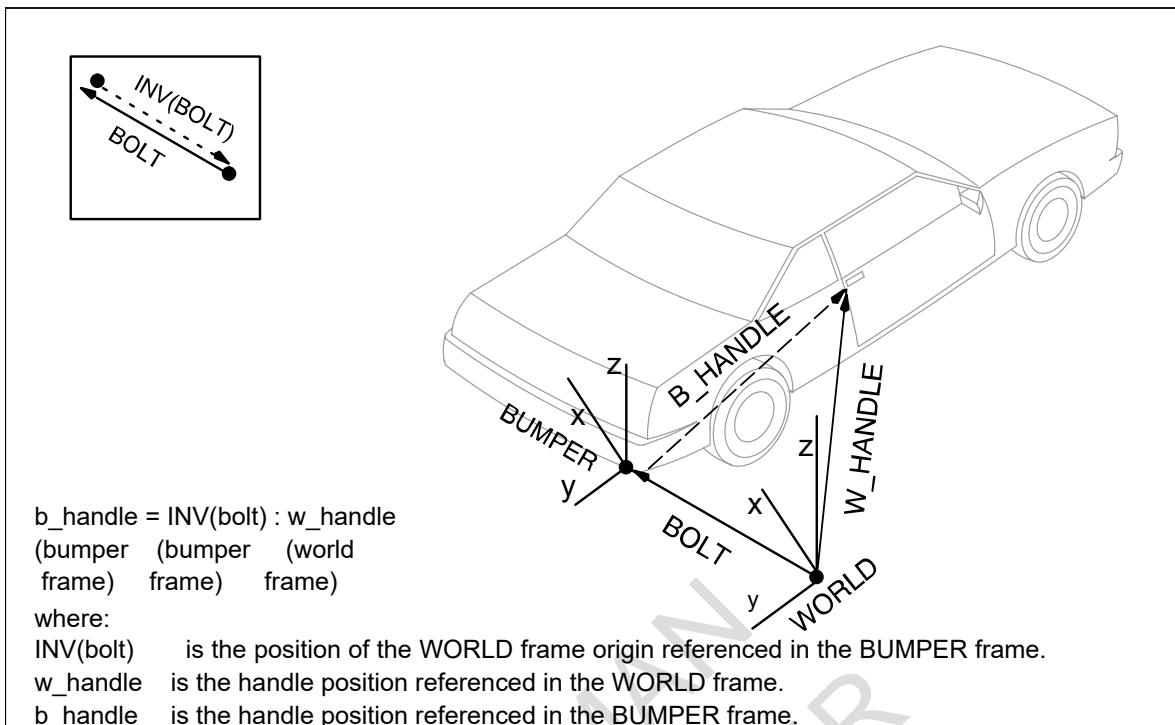
In the example shown in [Figure 3.2.4 \(b\)](#), a vision system is used to locate a target on a car such as a bolt head on a bumper. The relative position operator is used to calculate the position of the door handle based on data from the car drawings. The equation shown in [Figure 3.2.4 \(b\)](#) is used to calculate the position of *w\_handle* in the WORLD frame.



**Figure 3.2.4 (b) Determining w\_handle Relative to WORLD Frame**

The KAREL `INV` built-in function reverses the direction of the reference.

For example, to determine the position of the door handle target (`b_handle`) relative to the position of the bolt, use the equation shown in [Figure 3.2.4 \(c\)](#).



**Figure 3.2.4 (c) Determining `b_handle` Relative to BUMPER Frame**

#### NOTE

The order of the relative operator (`:`) is important, where: `b_handle=bolt:w_handle` **is not the same as** `b_handle=w_handle:bolt`

See Also: [Chapter 8, POSITION DATA](#), [Section A.9.17, INV Built-In Function](#)

#### Vector Operations

The following rules apply to VECTOR operations:

- A VECTOR expression can perform addition (+) and subtraction (-) equations on VECTOR operands. The result is a VECTOR whose components are the sum or difference of the corresponding components of the operands. For example, the components of the VECTOR `vect_3` will equal (5, 10, 9) as a result of the following program statements:

```
vect_1.x = 4; vect_1.y = 8; vect_1.z = 5
vect_2.x = 1; vect_2.y = 2; vect_2.z = 4
vect_3 = vect_1 + vect_2
```

**Figure 3.2.4 (d) Vector Operations**

- The multiplication (\*) and division (/) operators can be used with either
  - A VECTOR and an INTEGER operand
  - A VECTOR and a REAL operand

The product of a VECTOR and an INTEGER or a VECTOR and a REAL is a scaled version of the VECTOR. Each component of the VECTOR is multiplied by the INTEGER (treated as a REAL number) or the REAL.

For example, the VECTOR (8, 16, 10) is produced as a result of the following operation:

```
(4, 8, 5) * 2
```

VECTOR components can be on the left or right side of the operator.

- A VECTOR divided by an INTEGER or a REAL causes each component of the VECTOR to be divided by the INTEGER (treated as a REAL number) or REAL. For example,  $(4, 8, 5) / 2$  results in (2, 4, 2.5).

If the divisor is zero, the program is aborted with the Divide by zero error.

- An INTEGER or REAL divided by a VECTOR causes the INTEGER (treated as a REAL number) or REAL to be multiplied by the reciprocal of each element of the VECTOR, thus producing a new VECTOR. For example,  $3.5 / \text{VEC}(7.0, 8.0, 9.0)$  results in (0.5, 0.4375, 0.38889).

If any of the elements of the VECTOR are zero, the program is aborted with the Divide by zero error.

- The cross product operator (#) produces a VECTOR that is normal to the two operands in the direction indicated by the right hand rule and with a magnitude equal to the product of the magnitudes of the two vectors and  $\text{SIN}(\Theta)$ , where  $\Theta$  is the angle between the two vectors. For example,  $\text{VEC}(3.0, 4.0, 5.0) \# \text{VEC}(6.0, 7.0, 8.0)$  results in (-3.0, 6.0, -3.0).

If either vector is zero, or the vectors are exactly parallel, an error occurs.

- The inner product operator (@) results in a REAL number that is the sum of the products of the corresponding elements of the two vectors. For example,  $\text{VEC}(3.0, 4.0, 5.0) @ \text{VEC}(6.0, 7.0, 8.0)$  results in 86.0.
- If the result of any of the above operations is a component of a VECTOR with a magnitude too large for a KAREL REAL number, the program is aborted with the Real overflow error.

[Table 3.2.4](#) lists additional examples of vector operations.

**Table 3.2.4 Examples of Vector Operations**

EXPRESSION	RESULT
$\text{VEC}(3.0, 7.0, 6.0) + \text{VEC}(12.6, 3.2, 7.5)$	(15.6, 10.2, 13.5)
$\text{VEC}(7.6, 9.0, 7.0) - \text{VEC}(14.0, 3.5, 17.0)$	(-6.4, 5.5, -10)
$4.5 * \text{VEC}(3.2, 7.6, 4.0)$	(14.4, 34.2, 18.0)
$\text{VEC}(12.7, 2.0, 8.3) * 7.6$	(96.52, 15.2, 63.08)
$\text{VEC}(17.3, 1.5, 0.23) / 2$	(8.65, 0.75, 0.115)

## String Operations

The following rules apply to STRING operations:

- You can specify that a KAREL routine returns a STRING as its value. See [Figure 3.2.4 \(e\)](#).

```
ROUTINE name(parameter_list): STRING
    declares name as returning a STRING value
```

**Figure 3.2.4 (e) Specifying a KAREL Routine to Return a STRING Value**

- An operator can be used between strings to indicate the concatenation of the strings. See [Figure 3.2.4 \(f\)](#).

```
string_1 = string_2 + string_3 + 'ABC' + 'DEF'
```

**Figure 3.2.4 (f) Using an Operator to Concatenate Strings**

- STRING expressions can be used in WRITE statements. See [Figure 3.2.4 \(g\)](#).

```
WRITE(CHR(13) + string_1 + string_2)
      writes a single string consisting of a return
      character followed by string_1 and string_2
```

**Figure 3.2.4 (g) Using a STRING Expression in a WRITE Statement**

- During STRING assignment, the string will be truncated if the target string is not large enough to hold the same string.
- You can compare or extract a character from a string. For example if string\_1 = 'ABCDE', your output would be 'D'. See [Figure 3.2.4 \(h\)](#).

```
IF SUB_STR(string_1, 4, 1) = 'D' THEN
```

**Figure 3.2.4 (h) String Comparison**

- You can build a string from another string. See [Figure 3.2.4 \(i\)](#).

```
ROUTINE toupper(p_char: INTEGER): STRING
BEGIN
    IF (p_char > 96) AND (p_char < 123) THEN
        p_char = p_char - 32
    ENDIF
    RETURN (CHR(p_char))
END toupper
BEGIN
    WRITE OUTPUT ('Enter string: ')
    READ INPUT (string_1)
    string_2 = ''
    FOR idx = 1 TO STR_LEN(string_1) DO
        string_2 = string_2 + toupper(ORD(string_1, idx))
    ENDFOR
```

**Figure 3.2.4 (i) Building a String from Another String**

## 4 PROGRAM CONTROL

---

Program control structures define the flow of execution within a program or routine and include alternation, looping, and unconditional branching as well as execution control.

### 4.1 PROGRAM CONTROL STRUCTURES

---

Program control structures can be used to define the flow of execution within a program or routine. By default, execution starts with the first statement following the BEGIN statement and proceeds sequentially until the END statement (or a RETURN statement) is encountered. The following control structures are available in KAREL:

- Alternation
- Looping
- Unconditional Branching
- Execution Control
- Condition Handlers

For detailed information on each type of control structure, refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#).

#### 4.1.1 Alternation Control Structures

---

An alternation control structure allows you to include alternative sequences of statements in a program or routine. Each alternative can consist of several statements.

During program execution, an alternative is selected based on the value of one or more data items. Program execution then proceeds through the selected sequence of statements.

Two types of alternation control structures can be used:

- **IF Statement** - provides a means of specifying one of two alternatives based on the value of a BOOLEAN expression.
- **SELECT Statement** - used when a choice is to be made between several alternatives. An alternative is chosen depending on the value of the specified INTEGER expression.

**See Also:** [Section A.9.1, IF ... ENDIF Statement](#), [Section A.19.3, SELECT ... ENDSELECT Statement](#)

#### 4.1.2 Looping Control Statements

---

A looping control structure allows you to specify that a set of statements be repeated an arbitrary number of times, based on the value of data items in the program. KAREL supports three looping control structures:

- The **FOR statement** - used when a set of statements is to be executed a specified number of times. The number of times is determined by INTEGER data items in the FOR statement. At the beginning of

the FOR loop, the initial value in the range is assigned to an INTEGER counter variable. Each time the cycle is repeated, the counter is reevaluated.

- The **REPEAT statement** - allows execution of a sequence of statements to continue as long as some BOOLEAN expression remains FALSE. The sequence of executable statements within the REPEAT statement will always be executed once.
- The **WHILE statement** - used when an action is to be executed as long as a BOOLEAN expression remains TRUE. The boolean expression is tested at the start of each iteration, so it is possible for the action to be executed zero times.

See Also: [Section A.6.3, FOR...ENDFOR Statement](#) , [Section A.18.13, REPEAT ... UNTIL Statement](#) , [Section A.23.3, WHILE...ENDWHILE Statement](#)

### 4.1.3 Unconditional Branch Statement

Unconditional branching allows you to use a GO TO statement to transfer control from one place in a program to a specified label in another area of the program, without being dependent upon a condition or BOOLEAN expression.

 **WARNING**

Never include a GO TO statement into or out of a FOR loop. The program might be aborted with a Run time stack overflow error.

See Also: [Section A.7.28, GO TO Statement](#)

### 4.1.4 Execution Control Statements

The KAREL language provides the following program control statements, which are used to terminate or suspend program execution:

- **ABORT** - causes the execution of the program, including any motion in progress, to be terminated. The program cannot be continued after being aborted.
- **DELAY** - causes execution to be suspended for a specified time, expressed in milliseconds.
- **PAUSE** - causes execution to be suspended until a CONTINUE operation is executed.
- **WAIT FOR** - causes execution to be suspended until a specified condition or list of conditions is satisfied.

See Also: [Section A.1.3, ABORT Statement](#) , [Section A.4.11, DELAY Statement](#) , [Section A.16.5, PAUSE Statement](#) , [Section A.23.1, WAIT FOR Statement](#) , [Chapter 6, CONDITION HANDLERS](#)

### 4.1.5 Condition Handlers

A condition handler defines a series of actions which are to be performed whenever a specified condition is satisfied. Once defined, a condition handler can be ENABLED or DISABLED. Refer to [Chapter 6, CONDITION HANDLERS](#) for more information.

## 5 ROUTINES

---

Routines, similar in structure to a program, provide a method of modularizing KAREL programs. Routines can include VAR and/or CONST declarations and executable statements. Unlike programs, however, a routine must be declared within an upper case program, and cannot include other routine declarations.

KAREL supports two types of routines:

- Procedure Routines - do not return a value
- Function Routines - return a value

KAREL routines can be predefined routines called built-in routines or they can be user-defined.

The following rules apply to all KAREL routines:

- Parameters can be included in the declaration of a routine. This allows you to pass data to the routine at the time it is called, and return the results to the calling program.
- Routines can be called or invoked:
  - By the program in which they are declared
  - By any routine contained in that program
  - With declarations by another program, refer to [Section 5.1.1, Declaring Routines](#).

## 5.1 ROUTINE EXECUTION

---

This section explains the execution of procedure and function routines:

- Declaring routines
- Invoking routines
- Returning from routines
- Scope of variables
- Parameters and arguments

### 5.1.1 Declaring Routines

---

The following rules apply to routine declarations:

- A routine cannot be declared in another routine.
- The ROUTINE statement is used to declare both procedure and function routines.
- Both procedure and function routines must be declared before they are called.
- Routines that are local to the program are completely defined in the program. Declarations of local routines include:
  - The ROUTINE statement
  - Any VAR and/or CONST declarations for the routine
  - The executable statements of the routine
- While the VAR and CONST sections in a routine are identical in syntax to those in a program, the following restrictions apply:
  - PATH, FILE, and vision data types cannot be specified.

- FROM clauses are not allowed.
- IN clauses are not allowed.
- Routines that are local to the program can be defined after the executable section if the routine is declared using a FROM clause with the same program name. The parameters should only be defined once. See [Figure 5.1.1 \(a\)](#).

```
PROGRAM funct_lib
  ROUTINE done_yet(x: REAL; s1, s2: STRING): BOOLEAN FROM funct_lib
BEGIN
  IF done_yet(3.2, 'T', '')
  --
END funct_lib
ROUTINE done_yet
BEGIN
  --
END done_yet
```

**Figure 5.1.1 (a) Defining Local Routines Using a FROM Clause**

- Routines that are external to the program are declared in one program but defined in another.
  - Declarations of external routines include only the ROUTINE statement and a FROM clause.
  - The FROM clause identifies the name of the program in which the routine is defined.
  - The routine must be defined local to the program named in the FROM clause.
- You can include a list of parameters in the declaration of a routine. A parameter list is an optional part of the ROUTINE statement.
- If a routine is external to the program, the names in the parameter list are of no significance but must be included to specify the parameters. If there are no parameters, the parentheses used to enclose the list must be omitted for both external and local routines.

The examples in [Figure 5.1.1 \(b\)](#) illustrate local and external procedure routine declarations.

```
PROGRAM procs_lib
ROUTINE wait_a_bit
  --local procedure, no parameters
  BEGIN
    DELAY 20
  END wait_a_bit
ROUTINE toggle_out(i: INTEGER)
  --local procedure, one parameter
  BEGIN
    DOUT[i] = ON      --reference to parameter i
    DELAY 1000
    DOUT[i] = OFF
  END toggle_out
ROUTINE calc_dist(p1,p2: POSITION; dist: REAL) & FROM math_lib
  --external procedure defined in math_lib.kL
BEGIN
END procs_lib
```

**Figure 5.1.1 (b) Local and External Procedure Declarations**

The example in [Figure 5.1.1 \(c\)](#) illustrates local and external function routine declarations.

```
PROGRAM funct_lib
ROUTINE done_yet(x: REAL; s1, s2 :STRING): BOOLEAN&           FROM bool_lib
  --external function routine defined in bool_lib.kL
  --returns a BOOLEAN value
```

```

ROUTINE xy_dist(x1,y1,x2,y2: REAL): REAL
  --local function, returns a REAL value
  VAR
    sum_square: REAL  --dynamic local variable
    dx,dy: REAL       --dynamic local variables
  BEGIN
    dx = x2-x1      --references parameters x2 and x1
    dy = y2-y1      --references parameters y2 and y1
    sum_square = dx * dx + dy * dy
    RETURN(SQRT(sum_square))  --SQRT is a built-in
  END xy_dist
BEGIN
END funct_lib

```

**Figure 5.1.1 (c) Function Declarations**

See Also: [Section A.6.8, FROM Clause](#), [Section A.18.31, ROUTINE Statement](#)

## 5.1.2 Invoking Routines

Routines that are declared in a program can be called within the executable section of the program, or within the executable section of any routine contained in the program. Calling a routine causes the routine to be invoked. A routine is invoked according to the following procedure:

- When a routine is invoked, control of execution passes to the routine.
- After execution of a procedure is finished, control returns to the next statement after the point where the procedure was called.
- After execution of a function is finished, control returns to the assignment statement where the function was called.

The following rules apply when invoking procedure and function routines:

- Procedure and function routines are both called with the routine name followed by an argument for each parameter that has been declared for the routine.
- The argument list is enclosed in parentheses.
- Routines without parameters are called with only the routine name.
- A procedure is invoked as though it were a statement. Consequently, a procedure call constitutes a complete executable statement.

[Figure 5.1.2 \(a\)](#) shows the declarations for two procedures followed by the procedure calls to invoke them.

```

ROUTINE wait_a_bit FROM proc_lib
  --external procedure with no parameters
ROUTINE calc_dist(p1,p2: POSITION; dist: REAL) &
FROM math_lib
  --external procedure with three parameters
BEGIN
  ...
  wait_a_bit --invokes wait_a_bit procedure
  calc_dist (start_pos, end_pos, distance)
  --invokes calc_dist using three arguments for
  --the three declared parameters

```

**Figure 5.1.2 (a) Procedure Calls**

- Because a function returns a value, a function call must appear as part or all of an expression.
- When control returns to the calling program or routine, execution of the statement containing the function call is resumed using the returned value.

Figure 5.1.2 (b) shows the declarations for two functions followed by the function calls to invoke them.

```

ROUTINE error_check : BOOLEAN FROM error_prog
--external function with no parameters returns a BOOLEAN value
ROUTINE distance(p1, p2: POSITION) : REAL &
FROM funct_lib
--external function with two parameters returns a REAL value
BEGIN --Main program
--the function error_check is invoked and returns a BOOLEAN
--expression in the IF statement
IF error_check THEN
...
ENDIF
travel_time = distance(prev_pos, next_pos)/current_spd
--the function distance is invoked as part of an expression in
--an assignment statement

```

**Figure 5.1.2 (b) Function Calls**

- Routines can call other routines as long as the other routine is declared in the program containing the initial routine. For example, if a program named master\_prog contains a routine named call\_proc, that routine can call any routine that is declared in the program, master\_prog.
- A routine that calls itself is said to be recursive and is allowed in KAREL. For example, the routine factorial, shown in Figure 5.1.2 (c), calls itself to calculate a factorial value.

```

ROUTINE factorial(n: INTEGER) : INTEGER
--calculates the factorial value of the integer n
BEGIN
IF n = 0 THEN RETURN (1)
ELSE RETURN (n * factorial(n-1))
--recursive call to factorial
ENDIF
END factorial

```

**Figure 5.1.2 (c) Recursive Function**

- The only constraint on the depth of routine calling is the use of the KAREL *stack*, an area used for storage of temporary and local variables and for parameters. Routine calls cause information to be placed in memory on the stack. When the RETURN or END statement is executed in the routine, this information is taken off of the stack. If too many routine calls are made without this information being removed from the stack, the program will run out of stack space.

See Also: [Section 5.1.6, Stack Usage](#) for information on how much space is used on the stack for routine calls

## 5.1.3 Returning from Routines

The RETURN statement is used in a routine to restore execution control from a routine to the calling routine or program.

The following rules apply when returning from a routine:

- In a procedure, the RETURN statement cannot include a value.
- If no RETURN statement is executed, the END statement restores control to the calling program or routine.

Figure 5.1.3 (a) illustrates some examples of using the RETURN statement in a procedure.

```

ROUTINE gun_on (error_flag: INTEGER)
  --performs some operation while a "gun" is turned on
  --returns from different statements depending on what,
  --if any, error occurs.
VAR gun: INTEGER
BEGIN
  IF error_flag = 1 THEN RETURN
  --abnormal exit from routine, returns before
  --executing WHILE loop
  ENDIF
  WHILE DIN[gun] DO
  --continues until gun is off
  ...
  IF error_flag = 2 THEN RETURN
  --abnormal exit from routine, returns from
  --within WHILE loop
  ENDIF
ENDWHILE --gun is off
END gun_on --normal exit from routine

```

**Figure 5.1.3 (a) Procedure RETURN Statements**

- In a function, the RETURN statement must specify a value to be passed back when control is restored to the calling routine or program.
- The function routine can return any data type except
  - FILE
  - PATH
  - Vision types
- If the return type is an ARRAY, you cannot specify a size. This allows an ARRAY of any length to be returned by the function. The returned ARRAY, from an ARRAY valued function, can be used only in a direct assignment statement. ARRAY valued functions cannot be used as parameters to other routines. Refer to Figure 5.1.5 (c), for an example of an ARRAY passed between two function routines.
- If no value is provided in the RETURN statement of a function, a translator error is generated.
- If no RETURN statement is executed in a function, execution of the function terminates when the END statement is reached. No value can be passed back to the calling routine or program, so the program aborts with an error.

Figure 5.1.3 (b) illustrates some examples using the RETURN statement in function routines.

```

ROUTINE index_value (table: ARRAY of INTEGER;
  table_size: INTEGER): INTEGER
  --Returns index value of FOR loop (i) depending on
  --condition of IF statement. Returns 0 in cases where
  --IF condition is not satisfied.
VAR i: INTEGER
BEGIN
  FOR i = 1 TO table_size DO
    IF table[i] = 0 THEN RETURN (i) --returns index
    ENDIF
  ENDFOR

```

```

    RETURN (0) --returns 0
END index_value
ROUTINE compare (test_var_1: INTEGER;
    test_var_2: INTEGER): BOOLEAN
--Returns TRUE value in cases where IF test is
--satisfied. Otherwise, returns FALSE value.
BEGIN
    IF test_var_1 = test_var_2 THEN
        RETURN (TRUE) --returns TRUE
    ELSE
        RETURN (FALSE) --returns FALSE
    ENDIF
END compare

```

**Figure 5.1.3 (b) Function RETURN Statements**

See Also: [Section A.18.31, ROUTINE Statement](#)

## 5.1.4 Scope of Variables

---

The scope of a variable declaration can be

- Global
- Local

### Global Declarations and Definitions

The following rules apply to global declarations and definitions:

- Global declarations are recognized throughout a program.
- Global declarations are referred to as *static* because they are given a memory location that does not change during program execution, even if the program is cleared or reloaded (unless the variables themselves are cleared.)
- Declarations made in the main program, as well as predefined identifiers, are global.
- The scope rules for predefined and user-defined routines, types, variables, constants, and labels are as follows:
  - All predefined identifiers are recognized throughout the entire program.
  - Routines, types, variables, and constants declared in the declaration section of a program are recognized throughout the entire program, including routines that are in the program.

### Local Declarations and Definitions

The following rules apply to local declarations and definitions:

- Local declarations are recognized only within the routines where they are declared.
- Local data is created when a routine is invoked. Local data is destroyed when the routine finishes executing and returns.
- The scope rules for predefined and user-defined routines, variables, constants, and labels are as follows:
  - Variables and constants, declared in the declaration section of a routine, and parameters, declared in the routine parameter list, are recognized only in that routine.
  - Labels defined in a program (not in a routine of the program) are local to the body of the program and are not recognized within any routines of the program.

- Labels defined in a routine are local to the routine and are recognized only in that routine.
- Types cannot be declared in a routine, so are never local.

## 5.1.5 Parameters and Arguments

Identifiers that are used in the parameter list of a routine declaration are referred to as parameters. A parameter declared in a routine can be referenced throughout the routine. Parameters are used to pass data between the calling program and the routine. The data supplied in a call, referred to as arguments, can affect the way in which the routine is executed.

The following rules apply to the parameter list of a routine call:

- As part of the routine call, you must supply a data item, referred to as an *argument*, for each parameter in the routine declaration.
- An argument can be a variable, constant, or expression. There must be one argument corresponding to each parameter.
- Arguments must be of the same data type as the parameters to which they correspond, with three exceptions:
  - An INTEGER argument can be passed to a REAL parameter. In this case, the INTEGER value is treated as type REAL, and the REAL equivalent of the INTEGER is passed by value to the routine.
  - A BYTE or SHORT argument can be passed by value to an INTEGER or REAL parameter.
  - Any positional types can be passed to any other positional type. If they are being passed to a user-defined routine, the argument positional type is converted and passed by value to the parameter type.
  - ARRAY or STRING arguments of any length can be passed to parameters of the same data type.

Figure 5.1.5 (a) shows an example of a routine declaration and three calls to that routine.

**Figure 5.1.5 (a) Corresponding Parameters and Arguments**

```

PROGRAM params
  VAR
    long_string: STRING[10]; short_string: STRING[5]
    exact_dist: REAL; rough_dist: INTEGER
  ROUTINE label_dist (strg: STRING; dist: REAL) &
    FROM procs_lib
  BEGIN
    ...
    label_dist(long_string, exact_dist)
      --long_string corresponds to strg;
      --exact_dist corresponds to dist
    label_dist(short_string, rough_dist)
      --short_string, of a different length,
      --corresponds to strg; rough_dist, an
      --INTEGER, corresponds to REAL dist
    label_dist('new distance', (exact_dist * .75))
      --literal constant and REAL expression
      --arguments correspond to the parameters
  END params

```

- When the routine is invoked, the argument used in the routine call is passed to the corresponding parameter. Two methods are used for passing arguments to parameters:
  - **Passing Arguments By Reference**

If an argument is passed by reference, the corresponding parameter shares the same memory location as the argument. Therefore, changing the value of the parameter changes the value of the corresponding argument.

- **Passing Arguments By Value**

If an argument is passed by value, a temporary copy of the argument is passed to the routine. The corresponding parameter uses this temporary copy. Changing the parameter does not affect the original argument.

- Constant and expression arguments are always passed to the routine by value. Variables are normally passed by reference. The following variable arguments, however, are passed by value:
  - Port array variables
  - INTEGER variables passed to REAL parameters
  - BYTE and SHORT arguments passed to INTEGER or REAL parameters
  - System variables with read only (RO) access
  - Positional parameters that need to be converted
- While variable arguments are normally passed by reference, you can pass them by value by enclosing the variable identifier in parentheses. The parentheses, in effect, turn the variable into an expression.
- PATH, FILE, and vision variables can not be passed by value. ARRAY elements (indexed form of an ARRAY variable) can be passed by value, but entire ARRAY variables cannot.

[Figure 5.1.5 \(b\)](#) shows a routine that affects the argument being passed to it differently depending on how the variable argument is passed.

```

PROGRAM reference
  VAR arg : INTEGER
  ROUTINE test(param : INTEGER)
  BEGIN
    param = param * 3
    WRITE ('value of param:', param, CR)
  END test
  BEGIN
    arg = 5
    test((arg))    --arg passed to param by value
    WRITE('value of arg:', arg, CR)
    test(arg)      --arg passed to param by reference
    WRITE('value of arg:', arg, CR)
  END reference

```

**Figure 5.1.5 (b) Passing Variable Arguments**

The output from the program in [Figure 5.1.5 \(b\)](#) is as follows:

```

value of param: 15
value of arg: 5
value of param: 15
value of arg: 15

```

If the routine calls from [Figure 5.1.5 \(b\)](#) were made in reverse order, first passing `arg` by reference using `test(arg)` and then passing it by value using `test ((arg))`, the output would be affected as follows:

```

value of param: 15
value of arg: 15
value of param: 45
value of arg: 15

```

- To pass a variable as a parameter to a KAREL routine you can use one of two methods:
  - You can specify the name of the variable in the parameter list. For example, `other_rtn(param_var)` passes the variable `param_var` to the routine `other_rtn`. To write this statement, you have to know the name of the variable to be passed.
  - You can use BYNAME. The BYNAME feature allows a program to pass as a parameter to a routine a variable whose name is contained in a string. For example, if the string variables `prog_name` and `var_name` contain the name of a program and variable the operator has entered, this variable is passed to a routine using this syntax:  
`other_rtn(BYNAME(prog_name, var_name, entry))`
- If a function routine returns an ARRAY, a call to this function cannot be used as an argument to another routine. If an incorrect pass is attempted, a translation error is detected.

Figure 5.1.5 (c) shows the correct use of an ARRAY passed between two function routines.

```
PROGRAM correct
  VAR a : ARRAY[8] of INTEGER
    ROUTINE rtn_ary : ARRAY of INTEGER FROM util_prog
    ROUTINE print_ary(arg : ARRAY of INTEGER)
      VAR i : INTEGER
    BEGIN
      FOR i = 1 to ARRAY_LEN(arg) DO
        WRITE(arg[i],cr)
      ENDFOR
    END print_ary
    BEGIN
      a = rtn_ary
      print_ary(a)
    END correct
```

**Figure 5.1.5 (c) Correct Passage of an ARRAY**

Figure 5.1.5 (d) shows the incorrect use of an ARRAY passed between two function routines.

```
PROGRAM wrong
  ROUTINE rtn_ary : ARRAY of INTEGER FROM util_prog
  ROUTINE print_ary(arg : ARRAY of INTEGER)
    VAR i : INTEGER
  BEGIN
    FOR i = 1 to ARRAY_LEN(arg) DO
      WRITE(arg[i],cr)
    ENDFOR
  END print_ary
  BEGIN
    a = rtn_ary
    print_ary(a)
  END wrong
```

**Figure 5.1.5 (d) Incorrect Passage of an ARRAY**

See Also: [Section A.1.19, ARRAY\\_LEN Built-In Function](#) , [Section A.19.51, STR\\_LEN Built-In Function](#) , [Appendix E, SYNTAX DIAGRAMS](#)

## 5.1.6 Stack Usage

When a program is executed, a stack of 300 words is allocated unless you specify a stack size. The stack is allocated from available user RAM.

Stack usage can be calculated as follows:

- Each call (or function reference) uses at least five words of stack.
- In addition, for each parameter and local variable in the routine, additional space on the stack is used, depending on the variable or parameter type as shown in [Table 5.1.6](#).

**Table 5.1.6 Stack Usage**

Type	Parameter Passed by Reference	Parameter Passed by Value	Local Variable
BOOLEAN	1	2	1
ARRAY OF BOOLEAN		not allowed	1 + array size
ARRAY OF BYTE	1	not allowed	1 + (array size)/4
CAM_SETUP	1	not allowed	not allowed
ARRAY OF CAM_SETUP		not allowed	not allowed
CONFIG	1	2	1
ARRAY OF CONFIG		not allowed	1 + array size
INTEGER	1	2	1
ARRAY OF INTEGER		not allowed	1 + array size
FILE	1	not allowed	not allowed
ARRAY OF FILE		not allowed	not allowed
JOINTPOS	2	12	10
ARRAY OF JOINTPOS	1	not allowed	1 + 10 * array size
JOINTPOS1	2	4	2
ARRAY OF JOINTPOS1	1	not allowed	1 + 2 * array size
JOINTPOS2	2	5	3
ARRAY OF JOINTPOS2	1	not allowed	1 + 3 * array size
JOINTPOS3	2	6	4
ARRAY OF JOINTPOS3	1	not allowed	1 + 4 * array size
JOINTPOS4	2	7	5
ARRAY OF JOINTPOS4	1	not allowed	1 + 5 * array size
JOINTPOS5	2	8	6
ARRAY OF JOINTPOS5	1	not allowed	1 + 6 * array size
JOINTPOS6	2	9	7
ARRAY OF JOINTPOS6	1	not allowed	1 + 7 * array size

Type	Parameter Passed by Reference	Parameter Passed by Value	Local Variable
JOINTPOS7	2	10	8
ARRAY OF JOINTPOS7	1	not allowed	$1 + 8 * \text{array size}$
JOINTPOS8	2	11	9
ARRAY OF JOINTPOS8	1	not allowed	$1 + 9 * \text{array size}$
JOINTPOS9	2	12	10
ARRAY OF JOINTPOS9	1	not allowed	$1 + 10 * \text{array size}$
MODEL	1	not allowed	not allowed
ARRAY OF MODEL	1	not allowed	not allowed
PATH	2	not allowed	not allowed
POSITION	2	16	14
ARRAY OF POSITION	1	not allowed	$1 + 14 * \text{array size}$
REAL	1	2	1
ARRAY OF REAL	1	not allowed	$1 + \text{array size}$
ARRAY OF SHORT	1	not allowed	$1 + (\text{array size})/2$
STRING	2	$2 + (\text{string length}+2)/4$	$(\text{string length}+2)/4$
ARRAY OF STRING	1	not allowed	$1 + ((\text{string length}+2) * \text{array size})/4$
VECTOR	1	4	3
ARRAY OF VECTOR	1	not allowed	$1 + 3 * \text{array size}$
VIS_PROCESS	1	not allowed	not allowed
ARRAY OF VIS_PROCESS	1	not allowed	not allowed
XYZWPR	2	10	8
ARRAY OF XYZWPR	1	not allowed	$1 + 8 * \text{array size}$
XYZWPREXT	2	13	11
ARRAY OF XYZWPREX	1	not allowed	$1 + 11 * \text{array size}$
ARRAY [m,n] OF some_type	1	not allowed	$m(\text{ele size}/4 * n + 1)+1$
ARRAY [l,m,n] OF some_type	1	not allowed	$l(m(\text{ele size}/4 * n + 1)+1)+1$

## 5.2 BUILT-IN ROUTINES

---

The KAREL language includes predefined routines referred to as KAREL built-in routines, or built-ins. Predefined routines can be either procedure or function built-ins. They are provided as a programming convenience and perform commonly needed services.

Many of the built-ins return a status parameter that signifies an error if not equal to 0. The error returned can be any of the error codes defined in the *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)*. These errors can be posted to the error log and displayed on the error line by calling the POST\_ERR built-in routine with the returned status parameter.

[Table 5.2](#) is a summary list of all the predefined built-in routines included in the KAREL language. A detailed description of all the KAREL built-in routines is provided in [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#).

**Table 5.2 KAREL Built-In Routine Summary**

Category	Identifier		
<b>Byname</b>	CALL_PROG CALL_PROGLIN	CURR_PROG FILE_LIST	PROG_LIST VAR_INFO VAR_LIST
<b>Data Acquisition</b>	DAQ_CHECKP DAQ_REGPIPE	DAQ_START DAQ_STOP	DAQ_UNREG DAQ_WRITE
<b>Data Transfer Between Robots over Ethernet</b>	RGET_PORTCMT RGET_PORTSIM RGET_PORTVAL RGET_PREGCMT RGET_REG RGET_REG_CMT RGET_SREGCMT	RGET_STR_REG RNREG_RECV RNREG_SEND RPREG_RECV RPREG_SEND RSET_INT_REG RSET_PORTCMT	RSET_PORTSIM RSET_PORTVAL RSET_PREGCMT RSET_REALREG RSET_REG_CMT RSET_SREGCMT RSET_STR_REG
<b>Error Code Handling</b>	ERR_DATA	POST_ERR	POST_ERR_L
<b>File and Device Operation</b>	CHECK_NAME COMPARE_FILE COPY_FILE DELETE_FILE DISMOUNT_DEV DOSFILE_INF	FORMAT_DEV MOUNT_DEV MOVE_FILE PRINT_FILE PURGE_DEV RENAME_FILE	XML_ADDTAG XML_GETDATA XML_REMTAG XML_SCAN XML_SETVAR
<b>iPhone Communications</b>	RMCN_ALERT	RMCN_SEND	
<b>KCL Operation</b>	KCL	KCL_NO_WAIT	KCL_STATUS

Category	Identifier		
<b>Memory Operation</b>	CLEAR CREATE_VAR LOAD LOAD_STATUS	PROG_BACKUP PROG_CLEAR PROG_RESTORE RENAME_VAR	RENAME_VARS SAVE SAVE_DRAM
<b>Mirror</b>	MIRROR		
<b>Motion and Program Control</b>	CNCL_STP_MTN	MOTION_CTL	RESET
<b>Multi-programming</b>	ABORT_TASK CLEAR_SEMA CONT_TASK GET_TSK_INFO LOCK_GROUP	PAUSE_TASK PEND_SEMA POST_SEMA RUN_TASK	SEMA_COUNT SET_TSK_ATTR SET_TSK_NAME UNLOCK_GROUP
<b>Path Operation</b>	APPEND_NODE COPY_PATH	DELETE_NODE INSERT_NODE	NODE_SIZE PATH_LEN
<b>Personal Computer Communications</b>	ADD_BYNAMEPC ADD_INTPC	ADD_REALPC ADD_STRINGPC	SEND_DATAPC SEND_EVENTPC
<b>Position</b>	CHECK_EPOS CNV_JPOS_REL CNV_REL_JPOS CURPOS CURJPOS	FRAME IN_RANGE J_IN_RANGE JOINT2POS	POS POS2JOINT SET_PERCH UNPOS
<b>Process I/O Setup</b>	CLR_PORT_SIM GET_PORT_ASG GET_PORT_CMT GET_PORT_MOD	GET_PORT_SIM GET_PORT_VAL IO_MOD_TYPE SET_PORT_ASG	SET_PORT_CMT SET_PORT_MOD SET_PORT_SIM SET_PORT_VAL
<b>Queue Manager</b>	APPEND_QUEUE COPY_QUEUE DELETE_QUEUE	GET_QUEUE INIT_QUEUE	INSERT_QUEUE MODIFY_QUEUE
<b>Register Operation</b>	CLR_POS_REG GET_JPOS_REG GET_POS_REG GET_PREG_CMT GET_REG GET_REG_CMT	GET_SREG_CMT GET_STR_REG POS_REG_TYPE SET_EPOS_REG SET_INT_REG SET_JPOS_REG	SET_POS_REG SET_PREG_CMT SET_REAL_REG SET_REG_CMT SET_SREG_CMT SET_STR_REG

Category	Identifier		
<b>Serial I/O, File Usage</b>	BYTES_AHEAD BYTES_LEFT CLR_IO_STAT GET_FILE_POS GET_PORT_ATR	IO_STATUS MSG_CONNECT MSG_DISCO MSG_PING PIPE_CONFIG	SET_FILE_ATR SET_FILE_POS SET_PORT_ATR VOL_SPACE
<b>String Operation</b>	CNV_CNF_STRG CNV_CONF_STR CNV_INT_STR	CNV_REAL_STR CNV_STR_CONF CNV_STR_INT	CNV_STR_REAL STR_LEN SUB_STR
<b>System</b>	ABS ACOS ARRAY_LEN ASIN ATAN2 BYNAME CHR	COS EXP GET_VAR INDEX INV LN ORD	ROUND SET_VAR SIN SQRT TAN TRUNC UNINIT
<b>Time-of-Day Operation</b>	CNV_STR_TIME CNV_TIME_STR	GET_TIME GET_USEC_SUB	GET_USEC_TIM SET_TIME
<b>TPE Program</b>	AVL_POS_NUM CLOSE_TPE COPY_TPE CREATE_TPE DEL_INST_TPE GET_ATTR_PRG GET_JPOS_TPE	GET_POS_FRM GET_POS_TPE GET_POS_TYP GET_TPE_CMT GET_TPE_PRM OPEN_TPE SELECT_TPE	SET_ATTR_PRG SET_EPOS_TPE SET_JPOS_TPE SET_POS_TPE SET_TPE_CMT SET_TRNS_TPE
<b>Translate</b>	TRANSLATE		

Category	Identifier		
<b>User Interface</b>	ACT_SCREEN	DET_WINDOW	INI_DYN_DISS
	ADD_DICT	DISCTRL_ALPH	INIT_TBL
	ACT_TBL	DISCTRL_FORM	POP_KEY_RD
	ATT_WINDOW_D	DISCTRL_LIST	PUSH_KEY_RD
	ATT_WINDOW_S	DISCTRL_PLMN	READ_DICT
	CHECK_DICT	DISCTRL_SBMN	READ_DICT_V
	CNC_DYN_DISB	DISCTRL_TBL	READ_KB
	CNC_DYN_DISE	FORCE_LINK	REMOVE_DICT
	CNC_DYN_DISI	FORCE_SPMENU	SET_CURSOR
	CNC_DYN_DISP	INI_DYN_DISB	SET_LANG
	CNC_DYN_DISR	INI_DYN_DISE	WRITE_DICT
	CNC_DYN_DISS	INI_DYN_DISI	WRITE_DICT_V
	DEF_SCREEN	INI_DYN_DISP	
	DEF_WINDOW	INI_DYN_DISR	
<b>Vector</b>	APPROACH	ORIENT	
<b>Vision Operation</b>	V_ACQ_VAMAP	V_GET_OFFSET	VT_CLR_QUEUE
	V_ADJ_2D	V_GET_PASSFL	VT_DELETE_PQ
	V_CAM_CALIB	V_GET_READ	VT_GET_AREID
	V_CAM_CHECK	V_GET_VPARAM	VT_GET_FOUND
	V_CLR_VAMAP	V_IRCONNECT	VT_GET_LINID
	V_CSAPI_GETVALUE	V_LED_OFF	VT_GET_PFRT
	V_CSAPI_NUMSET	V_LED_ON	VT_GET_QUEUE
	V_CSAPI_RESETDATA	V_OVERRIDE	VT_GET_TIME
	V_CSAPI_SAVEDATA	V_RUN_FIND	VT_GET_TRYID
	V_CSAPI_SETVALUE	V_SAVE_IMREG	VT_PUT_QUE2
	V_CSAPI_TESTRUN	V_SET_REF	VT_READ_PQ
	V_DISPLAY4D	V_SNAP_VIEW	VT_SET_FLAG
	V_FIND_VIEW	VREG_FND_POS	VT_SET_LDBAL
	V_FIND_VLINE	VREG_OFFSET	VT_WRITE_PQ
	V_GET_FOUND	VT_ACK_QUEUE	

EFFMAN  
QUIRIONR

## 6 CONDITION HANDLERS

---

The condition handler feature of the KAREL language allows a program to respond to external conditions more efficiently than conventional program control structures allow.

These condition handlers, also known as Global condition handlers, allow specified conditions to be monitored in parallel with normal program execution and, if the conditions occur, corresponding actions to be taken in response.

For a condition handler to be monitored, it must be defined first and then enabled. Disabling a condition handler removes it from the group being scanned. Purging condition handlers deletes their definition.

[Table 6 \(a\)](#) lists the conditions that can be monitored by condition handlers.

**Table 6 (a) Conditions**

ITEM	DESCRIPTION
port_id[n]	ERROR[n]
NOT port_id[n]	EVENT[n]
port_id[n]+	ABORT
port_id[n]-	PAUSE
operand = operand	CONTINUE
operand <> operand	SEMAPHORE[n]
operand < operand	
operand <= operand	
operand > operand	
operand >= operand	

[Table 6 \(b\)](#) lists the actions that can be taken.

**Table 6 (b) Actions**

variable = expression	NOABORT
port_id[n] = expression	NOMESSAGE
STOP	NOPAUSE
CANCEL	ENABLE CONDITION[n]
RESUME	DISABLE CONDITION[n]
HOLD	PULSE DOUT[n] FOR t
UNHOLD	UNPAUSE
routine_name	ABORT
SIGNAL EVENT[n]	CONTINUE
	PAUSE
	SIGNAL SEMAPHORE[n]

## 6.1 CONDITION HANDLER OPERATIONS

---

[Table 6.1](#) summarizes condition handler operations.

**Table 6.1 Condition Handler Operations**

OPERATION	GLOBAL CONDITION HANDLER
Define	CONDITION[n]:<WITH \$SCAN_TIME = n> WHEN conditions DO actions ENDCONDITION
Enable	ENABLE CONDITION[n] (statement or action)
Disable	DISABLE CONDITION[n] (statement or action) or conditions satisfied
Purge	PURGE CONDITION[n] (statement), program terminated

### 6.1.1 Global Condition Handlers

---

Global condition handlers are defined by executing a CONDITION statement in the executable section of a program. The definition specifies conditions/actions pairs. The following rules apply to global condition handlers.

- Each global condition handler is referenced throughout the program by a specified number, from 1 to 1000. If a condition handler with the specified number was previously defined, it must be purged before it is replaced by the new one.
- The conditions/actions pairs of a global condition handler are specified in the WHEN clauses of a CONDITION statement. All WHEN clauses for a condition handler are enabled, disabled, and purged together.
- The condition list represents a list of conditions to be monitored when the condition handler is scanned.
- By default, each global condition handler is scanned at a rate based on the value of `$SCR.$cond_time`. If the `WITH $SCAN_TIME=n` clause is used in a CONDITION statement, the condition will be scanned roughly every n milliseconds. The actual interval between the scans is determined as shown in [Table 6.1.1](#).

**Table 6.1.1 Interval Between Global Condition Handler Scans**

n	Interval Between Scans
<code>n &lt;= \$COND_TIME</code>	<code>\$COND_TIME</code>
<code>\$COND_TIME &lt; n &lt;= (2 * \$COND_TIME)</code>	<code>(2 * \$COND_TIME)</code>
<code>(2 * \$COND_TIME) &lt; n &lt;= (4 * \$COND_TIME)</code>	<code>(4 * \$COND_TIME)</code>
<code>(4 * \$COND_TIME) &lt; n &lt;= (8 * \$COND_TIME)</code>	<code>(8 * \$COND_TIME)</code>
<code>(8 * \$COND_TIME) &lt; n &lt;= (16 * \$COND_TIME)</code>	<code>(16 * \$COND_TIME)</code>
<code>(16 * \$COND_TIME) &lt; n &lt;= (32 * \$COND_TIME)</code>	<code>(32 * \$COND_TIME)</code>
<code>(32 * \$COND_TIME) &lt; n &lt;= (64 * \$COND_TIME)</code>	<code>(64 * \$COND_TIME)</code>
<code>(64 * \$COND_TIME) &lt; n &lt;= (128 * \$COND_TIME)</code>	<code>(128 * \$COND_TIME)</code>

n	Interval Between Scans
$(128 * \$COND\_TIME) < n \leq (256 * \$COND\_TIME)$	$(256 * \$COND\_TIME)$
$(256 * \$COND\_TIME) < n$	$(512 * \$COND\_TIME)$

- Multiple conditions must all be separated by the AND operator or the OR operator. Mixing of AND and OR is not allowed.
- If AND is used, all of the conditions of a single WHEN clause must be satisfied simultaneously for the condition handler to be triggered.
- If OR is used, the actions are triggered when any of the conditions are TRUE.
- The action list represents a list of actions to be taken when the corresponding conditions of the WHEN clause are simultaneously satisfied.
- Multiple actions must be separated by a comma or a new line.

Figure 6.1.1 (a) shows three examples of defining global condition handlers.

```

CONDITION[1]:      --defines condition handler number
1
    WHEN DIN[1] DO DOUT[1] = TRUE          --triggered if any one
    WHEN DIN[2] DO DOUT[2] = TRUE          --of the WHEN clauses
    WHEN DIN[3] DO DOUT[3] = TRUE          --is satisfied
ENDCONDITION
CONDITION[2]:      --defines condition handler number 2
    WHEN PAUSE DO                         --one condition triggers
        AOUT[speed_out] = 0                --multiple actions
        DOUT[pause_light] = TRUE
    ENABLE CONDITION [2]                  --enables this condition
ENDCONDITION          --handler again
CONDITION[3]:
    WHEN DIN[1] AND DIN[2] AND DIN[3] DO --multiple
        DOUT[1] = TRUE                   --conditions separated by AND;
        DOUT[2] = TRUE                   --all three conditions must be
        DOUT[3] = TRUE                   --satisfied at the same time
ENDCONDITION

```

Figure 6.1.1 (a) Global Condition Handler Definitions

- You can enable, disable, and purge global condition handlers as needed throughout the program. Whenever a condition handler is triggered, it is automatically disabled, unless an ENABLE action is included in the action list. (See condition handler 2 in Figure 6.1.1 (a).)
  - The ENABLE statement or action enables the specified condition handler. The condition handler will be scanned during the next scan operation and will continue to be scanned until it is disabled.
  - The DISABLE statement or action removes the specified condition handler from the group of scanned condition handlers. The condition handler remains defined and can be enabled again with the ENABLE statement or action.
  - The PURGE statement deletes the definition of the specified condition handler.
- ENABLE, DISABLE, and PURGE have no effect if the specified condition handler is not defined. If the specified condition handler is already enabled, ENABLE has no effect; if it is already disabled, DISABLE has no effect.

Figure 6.1.1 (b) shows examples of enabling, disabling, and purging global condition handlers.

```

CONDITION[1]:      --defines condition handler number 1
    WHEN line_stop = TRUE DO DOUT[1] = FALSE
ENDCONDITION
CONDITION[2]:      --defines condition handler number 2

```

```

WHEN line_go = TRUE DO
    DOUT[1] = TRUE, ENABLE CONDITION [1]
ENDCONDITION
ENABLE CONDITION[2] --condition handler 2 is enabled
. .
IF ready THEN line_go = TRUE; ENDIF
--If ready is TRUE condition handler 2 is triggered (and
--disabled) and condition handler 1 is enabled.
--Otherwise, condition handler 2 is not triggered (and is
--still enabled), condition handler 1 is not yet enabled,
--and the next two statements will have no effect.
DISABLE CONDITION[1]
ENABLE CONDITION[2]
. .
ENABLE CONDITION[1] --condition handler 1 is enabled
. .
line_stop = TRUE --triggers (and disables) condition handler 1
. .
PURGE CONDITION[2] --definition of condition handler 2 deleted
ENABLE CONDITION[2] --no longer has any effect
line_go = TRUE --no longer a monitored condition

```

**Figure 6.1.1 (b) Using Global Condition Handlers**

## 6.2 CONDITIONS

---

One or more conditions are specified in the condition list of a WHEN or UNTIL clause, defining the conditions portion of a conditions/actions pair. Conditions can be

- States - which remain satisfied as long as the state exists. Examples of states are DIN[1] and (VAR1 > VAR2).
- Events - which are satisfied only at the instant the event occurs. Examples of events are ERROR[n], DIN[n]+, and PAUSE.

The following rules apply to system and program event conditions:

- After a condition handler is enabled, the specified conditions are monitored.
  - If all of the conditions of an AND, WHEN, or UNTIL clause are simultaneously satisfied, the condition handler is triggered and corresponding actions are performed.
  - If all of the conditions of an OR, WHEN, or UNTIL clause are satisfied, the condition handler is triggered and corresponding actions are performed.
- Event conditions very rarely occur simultaneously. Therefore, you should never use AND between two event conditions in a single WHEN or UNTIL clause because, both conditions will not be satisfied simultaneously.
- While many conditions are similar in form to BOOLEAN expressions in KAREL, and are similar in meaning, only the forms listed in this section, not general BOOLEAN expressions, are permitted.
- Expressions are permitted within an EVAL clause. More general expressions may be used on the right side of comparison conditions, by enclosing the expression in an EVAL clause: EVAL (expression). However, expressions in an EVAL clause are evaluated when the condition handler is defined. They are not evaluated dynamically.
- The value of an EVAL clause expression must be INTEGER, REAL, or BOOLEAN.

See Also: [Section A.5.6, EVAL Clause](#)

## 6.2.1 Port\_Id Conditions

Port\_id conditions are used to monitor digital port signals. Port\_id must be one of the predefined BOOLEAN port array identifiers (DIN, DOUT, OPIN, OPOUT, TPIN, TPOUT, RDI, RDO, WDI, or WDO). The value of n specifies the port array signal to be monitored. [Table 6.2.1](#) lists the available port\_id conditions.

**Table 6.2.1 Port\_Id Conditions**

CONDITION	SATISFIED (TRUE) WHEN
port_id[n]	Digital port n is TRUE. (state)
NOT port_id[n]	Digital port n is FALSE. (state)
port_id[n]+	Digital port n changes from FALSE to TRUE. (event)
port_id[n]-	Digital port n changes from TRUE to FALSE. (event)

- For the state conditions, `port_id[n]` and `NOT port_id[n]`, the port is tested during every scan. The following conditions would be satisfied if, during a scan, DIN[1] was TRUE and DIN[2] was FALSE:

```
WHEN DIN[1] AND NOT DIN[2] DO . . .
```

Note that an input signal should remain ON or OFF for the minimum scan time to ensure that its state is detected.

- For the event condition `port_id[n] +`, the initial port value is tested when the condition handler is enabled. Each scan tests for the specified change in the signal. The change must occur while the condition handler is enabled.

The following condition would only be satisfied if, while the condition handler was enabled, DIN[1] changed from TRUE to FALSE since the last scan.

```
WHEN DIN[1]- DO . . .
```

## 6.2.2 Relational Conditions

Relational conditions are used to test the relationship between two operands. They are satisfied when the specified relationship is TRUE. Relational conditions are state conditions, meaning the relationship is tested during every scan. [Table 6.2.2](#) lists the relational conditions.

**Table 6.2.2 Relational Conditions**

CONDITION	SATISFIED (TRUE) WHEN
operand = operand	Relationship specified is TRUE. Operands on the left can be a port array element, referenced as <code>port_id[n]</code> , or a variable.
operand < > operand	Operands on the right can be a variable, a constant, or an EVAL clause. (state)
operand < operand	
operand < = operand	
operand > operand	
operand > = operand	

The following rules apply to relational conditions:

- Both operands must be of the same data type and can only be of type INTEGER, REAL, or BOOLEAN. (As in other situations, INTEGER constants can be used where REAL values are required, and will be treated as REAL values.)
- The operand on the left side of the condition can be any of the port array signals, a user-defined variable, a static variable, or a system variable that can be read by a KAREL program.
- The operand on the right side of the condition can be a user-defined variable, a static variable, a system variable that can be read by a KAREL program, any constant, or an EVAL clause. For example:

```

WHEN DIN[1] = ON DO . . .          --port_id and constant
WHEN flag = TRUE DO . . .         --variable and constant
WHEN AIN[1] >= temp DO . . .      --port_id and variable
WHEN flag_1 <> flag_2 DO . . .    --variable and variable
WHEN AIN[1] <= EVAL(temp * scale) DO . . .
    --port_id and EVAL clause
WHEN dif > EVAL(max_count - count) DO . . .
    --variable and EVAL clause

```

- The EVAL clause allows you to include expressions in relational conditions. However, it is evaluated only when the condition handler is defined. The expression in the EVAL clause cannot include any routine calls.

#### [Section A.5.6, EVAL Clause](#)

### 6.2.3 System and Program Event Conditions

System and program event conditions are used to monitor system and program generated events. The specified condition is satisfied only if the event occurs when the condition handler is enabled.

Enabled condition handlers containing ERROR, EVENT, PAUSE, ABORT, POWERUP, or CONTINUE conditions are scanned only if the specified type of event occurs. For example, an enabled condition handler containing an ERROR condition will be scanned only when an error occurs. [Table 6.2.3](#) lists the available system and program event conditions.

**Table 6.2.3 System and Program Event Conditions**

CONDITION	SATISFIED (TRUE) WHEN
ERROR [n]	The error specified by n is reached or, if n = *, any error occurs. (event)
EVENT[n]	The event specified by n is signaled. (event)
ABORT	The program is aborted. (event)
PAUSE	The program is paused. (event)
CONTINUE	The program is continued. (event)
POWERUP	The program is continued. (event)
SEMAPHORE[n]	The value of the semaphore specified by n is posted.

The following sections describe the rules that apply to these conditions.

## ERROR Condition

- The ERROR condition can be used to monitor the occurrence of a particular error by specifying the error code for that error. For example, ERROR [15018] monitors the occurrence of the error represented by the error code 15018.

The error codes are listed in the following format:

ffccc (decimal)

where

ff represents the facility code of the error  
 ccc represents the error code within the specified facility

For example, 15018 is **MOTN-018**, which is Position not reachable. The facility code is 15 and the error code is 018. Refer to the *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)* for a complete listing of error codes.

- The ERROR condition can also be used to monitor the occurrence of any error by specifying an asterisk (\*), the wildcard character, in place of a specific error code. For example, ERROR [\*] monitors the occurrence of any error.
- The ERROR condition is satisfied only for the scan performed when the error was detected. The error is not remembered in subsequent scans.

## EVENT Condition

- The EVENT condition monitors the occurrence of the specified program event. The SIGNAL statement or action in a program indicates that an event has occurred.
- The EVENT condition is satisfied only for the scan performed when the event was signaled. The event is not remembered in subsequent scans.

## ABORT Condition

- The ABORT condition monitors the aborting of program execution. If an ABORT occurs, the corresponding actions are performed. However, if one of the actions is a routine call, the routine will not be executed because program execution has been aborted.
- If an ABORT condition is used in a condition handler all actions, except routine calls, will be performed even though the program has aborted.

## PAUSE Condition

- The PAUSE condition monitors the pausing of program execution. If one of the corresponding actions is a routine call, it is also necessary to specify a NOPAUSE or UNPAUSE action.

## CONTINUE Condition

- The CONTINUE condition monitors the resumption of program execution. If program execution is paused, the CONTINUE action, the KCL> CONTINUE command, a CYCLE START from the operator panel, or the teach pendant FWD key will continue program execution and satisfy the CONTINUE condition.

### **POWERUP Condition**

- The POWERUP condition monitors the resumption of program execution after a power failure recovery. The controller must be able to recover successfully from a power failure before the program can be resumed.

### **SEMAPHORE Condition**

- The SEMAPHORE condition monitors the specified semaphore. The CLEAR\_SEMA built-in can be used to set the semaphore value to 0. The POST\_SEMA built-in or the SIGNAL\_SEMAPHORE action can be used to increment the semaphore value and satisfy the SEMAPHORE condition.

See Also: [Section A.1.2, ABORT Condition](#) , [Section A.3.44, CONTINUE Condition](#) , [Section A.5.5, ERROR Condition](#) , [Section A.5.7, EVENT Condition](#) , [Section A.16.4, PAUSE Condition](#) , [Section A.19.6, SEMAPHORE Condition](#) , the *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)*

## **6.3 ACTIONS**

---

Actions are specified in the action list of a WHEN clause. Actions can be

- Specially defined KAREL actions that are executed in parallel with the program
- A routine call, which will interrupt program execution

When the conditions of a condition handler are satisfied, the condition handler is triggered. The actions corresponding to the satisfied conditions are performed in the sequence in which they appear in the condition handler definition, except for routine calls. Routines are executed after all of the other actions have been performed.

Note that, although many of the actions are similar in form to KAREL statements and the effects are similar to corresponding KAREL statements, the actions are not executable statements. Only the forms indicated in this section are permitted.

See Also: [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#)

### **6.3.1 Assignment Actions**

---

The available assignment actions are given in [Table 6.3.1](#) .

**Table 6.3.1 Assignment Actions**

ACTION	RESULT
variable = expression	The value of the expression is assigned to the variable. The expression can be a variable, a constant, a port array element, or an EVAL clause.
port_id[n] = expression	The value of the expression is assigned to the port array element referenced by n. The expression can be a variable, a constant, or an EVAL clause.

The following rules apply to assignment actions:

- The assignment actions, variable=expression and port\_id[n]=expression can be used to assign values to variables and port array elements.

- The variable must be either a user-defined variable, a static variable, or a system variable without a minimum/maximum range and that can be written to by a KAREL program.
- The port array, if on the left, must be an output port array that can be set by a KAREL program.
- The expression can be a user-defined variable, a static variable, a system variable that can be read by a KAREL program, any constant, or an EVAL clause.
- If a variable is on the left side of the assignment, the expression can also be a port array element. However, you cannot assign a port array element to a port array element directly. For example, the first assignment shown is invalid, but the next two are valid:

```

DOUT[1] = DOUT[2]    --invalid action
port_var = DOUT[2]   --valid action, where port_var is a
variable
DOUT[1] = port_var  --another valid action, which if executed
                     --after port_var = DOUT[2], would in effect
                     --assign DOUT[2] to DOUT[1]

```

- If the expression is a variable, it must be a global variable. The value used is the current value of the variable at the time the action is taken, not when the condition handler is defined. If the expression is an EVAL clause, it is evaluated when the condition handler is defined and that value is assigned when the action is taken.
- Both sides of the assignment action must be of the same data type. An INTEGER or EVAL clause is permitted on the right side of the assignment with an INTEGER, REAL, or BOOLEAN on the left.

### 6.3.2 Motion Related Actions

Motion related actions affect the current motion and might affect subsequent motions. They are given in [Table 6.3.2](#).

**Table 6.3.2 Motion Related Actions**

ACTION	RESULT
STOP	Current motion is stopped.
RESUME	The last stopped motion is resumed.
CANCEL	Current motion is canceled.
HOLD	Current motion is held. Subsequent motions are not started.
UNHOLD	Held motion is released.

The following rules apply to motion related actions:

- If a STOP is issued, the current motion and any queued motions are pushed as a set on a stopped motion stack. If no motion is in progress, an empty entry is pushed on the stack.
- If a RESUME is issued, the newest stopped motion set on the stopped motion stack is queued for execution.
- If a CANCEL is issued, the motion currently in progress is canceled. Any motions queued to the same group behind the current motion are also canceled. If no motion is in progress, the action has no effect.
- If a HOLD is issued, the current motion is held and subsequent motions are prevented from starting. The UNHOLD action releases held motion.

### 6.3.3 Routine Call Actions

Routine call actions, or interrupt routines, are specified by

```
<WITH $PRIORITY=n> routine_name
```

The following restrictions apply to routine call actions or interrupt routines:

- The interrupt routine cannot have parameters and must be a procedure (not a function).
- If the interrupted program is using READ statements, the interrupt routine cannot read from the same file variable. If an interrupted program is reading and the interrupt routine attempts a read from the same file variable, the program is aborted.
- When an interrupt routine is started, the interrupted KAREL program is suspended until the routine returns.
- Interrupt routines, like KAREL programs, can be interrupted by other routines. The maximum depth of interruption is limited only by stack memory size.
- Routines are started in the sequence in which they appear in the condition handler definition, but since they interrupt each other, they will actually execute in reverse order.
- Interrupts can be prioritized so that certain interrupt routines cannot be interrupted by others. The \$PRIORITY condition handler qualifier can be used to set the priority of execution for an indicated routine action. \$PRIORITY values must be 0-255 where the lower value represents a lower priority. If a low priority routine is called while a routine with a higher priority is running, it will be executed only when the higher priority routine has completed. If \$PRIORITY is not specified, the routine's priority will default to the current value of the \$PRIORITY system variable.

See Also: [Section A.23.4, WITH Clause](#), for more information on \$PRIORITY

### 6.3.4 Miscellaneous Actions

[Table 6.3.4](#) describes other allowable actions.

**Table 6.3.4 Miscellaneous Actions**

ACTION	RESULT
SIGNAL EVENT[n]	The event specified by n is signaled.
NOMESSAGE	The error message that otherwise would have been generated is not displayed or logged.
NOPAUSE	Program execution is resumed if the program was paused, or is prevented from pausing.
NOABORT	Program execution is resumed if the program was aborted, or is prevented from aborting.
ABORT	Program execution is aborted.
CONTINUE	Program execution is continued.
PAUSE	Program execution is paused.
SIGNAL SEMAPHORE[n]	Specified semaphore is signaled.
ENABLE CONDITION[n]	Condition handler n is enabled.
DISABLE CONDITION[n]	Condition handler n is disabled.

ACTION	RESULT
PULSE DOUT[n] FOR t	Specified port n is pulsed for the time interval t (in milliseconds).
UNPAUSE	If a routine_name is specified as an action, but program execution is paused, execution is resumed only for the duration of the routine and then is paused again.

See Also: [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more information on each miscellaneous action.

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# 7 FILE INPUT/OUTPUT OPERATIONS

---

The KAREL language facilities allow you to perform the following serial input/output (I/O) operations:

- Open data files and serial communication ports using the OPEN FILE statement
- Close data files and serial communication ports using the CLOSE FILE statement
- Read from files, communication ports, and user interface devices using the READ statement
- Write to files, communication ports, and user interface devices using the WRITE statement
- Cancel read or write operations

*File variables* are used to indicate the file, communication port, or device on which a serial I/O operation is to be performed.

*Buffers* are used to hold data that has not yet been transmitted. The use of data items in READ and WRITE statements and their format specifiers depend on whether the data is text (ASCII) or binary, and on the data type.

## 7.1 FILE VARIABLES

---

A KAREL program can perform serial I/O operations on the following:

- Data files residing in the KAREL file system
- Serial communication ports associated with connectors on the KAREL controller
- User interface devices including the CRT/KB and teach pendant

A file variable is used to indicate the file, communication port, or device on which you want to perform a particular serial I/O operation.

**Table 7.1** lists the predefined file variables for user interface devices. These file variables are already opened and can be used in the READ or WRITE statements.

**Table 7.1 Predefined File Variables**

IDENTIFIER	DEVICE	OPERATIONS
TPFUNC*	Teach pendant function key line	Both
TPDISPLAY*	Teach pendant KAREL display	Both
TPPROMPT*	Teach pendant prompt line	Both
TPERROR	Teach pendant message line	Write
TPSTATUS*	Teach pendant status line	Write
CRTFUNC*	CRT/KB function key line	Both
INPUT	CRT/KB keyboard	Read
OUTPUT*	CRT/KB KAREL screen	Write
CRTPROMPT*	CRT/KB prompt line	Both
CRTERROR	CRT/KB message line	Write
CRTSTATUS*	CRT/KB status line	Write

\* Only displayed when teach pendant or CRT is in the **USER** menu.

A file variable can be specified in a KAREL statement as a FILE variable. [Figure 7.1](#) shows an example of declaring a FILE variable and of using FILE in the executable section of a program.

```
PROGRAM lun_prog
  VAR
    curnt_file : FILE
  ROUTINE input_data(file_spec:FILE) FROM util_prog
BEGIN
  OPEN FILE curnt_file ('RW','text.dt') --variable FILE
  input_data(curnt_file) --file variable argument
  WRITE TPERROR ('Error has occurred')
END lun_prog
```

**Figure 7.1 Using FILE in a KAREL Program**

Sharing FILE variables between programs is allowed as long as a single task is executing the programs. Sharing file variables between tasks is not allowed.

## 7.2 OPEN FILE STATEMENT

The OPEN FILE statement associates the file variable with a particular data file or communication port.

The association remains in effect until the file is closed, either explicitly by a CLOSE FILE statement or implicitly when program execution terminates or is aborted.

The OPEN FILE statement specifies how the file is to be used (usage string), and which file or port (file string) is used.

### 7.2.1 Setting File and Port Attributes

Attributes specify the details of operation of a serial port, or KAREL FILE variable. The SET\_PORT\_ATR and SET\_FILE\_ATR built-ins are used to set these attributes. SET\_FILE\_ATR must be called before the FILE is opened. SET\_PORT\_ATR can be called before or after the FILE that is using a serial port, is opened.

[Table 7.2.1 \(a\)](#) lists each attribute type, its function and whether the attribute is intended for use with teach pendant and CRT/KB devices, serial ports, data files, or pipes. Refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more information.

**Table 7.2.1 (a) Predefined Attribute Types**

ATTRIBUTE TYPE	FUNCTION	SET_PORT_ATR OR SET_FILE_ATR	TP/ CRT	SERIAL PORTS	DATA FILES	PIPES	SOCKET MSG.
ATR_BAUD	Baud rate	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_DBITS	Data length	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_EOL	End of line	SET_FILE_ATR	not used	valid	not used	valid	valid

ATTRIBUTE TYPE	FUNCTION	SET_PORT_ATR OR SET_FILE_ATR	TP/ CRT	SERIAL PORTS	DATA FILES	PIPES	SOCKET MSG.
ATR_FIELD	Field	SET_FILE_ATR	valid	valid	valid	valid	valid
ATR_IA	Interactively write	SET_FILE_ATR	valid	valid	valid	valid	valid
ATR_MODEM	Modem line	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_PARITY	Parity	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_PASSALL	Passall	SET_FILE_ATR	valid	valid	not used	valid	valid
ATR_READAHD	Read ahead buffer	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_REVERSE	Reverse transfer	SET_FILE_ATR	not used	valid	valid	valid	valid
ATR_SBITS	Stop bits	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_TIMEOUT	Timeout	SET_FILE_ATR	valid	valid	not used	valid	valid
ATR_UF	Unformatted transfer	SET_FILE_ATR	not used	valid	valid	valid	valid
ATR_XONOFF	XON/XOFF	SET_PORT_ATR	not used	valid	not used	not used	not used
ATR_PIPOVADV	Pipe Overflow	SET_FILE_ATR	not used	not used	not used	valid	valid
ATR_PIPWAIT	Wait for data	SET_FILE_ATR	not used	not used	not used	valid	valid

Table 7.2.1 (b) contains detailed explanations of each attribute.

Table 7.2.1 (b) Attribute Values

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
ATR_BAUD Baud rate	The baud rate of a serial port can be changed to one of the valid attribute values.	PORT	Read/ Write	BAUD_9600: 9600 baud  BAUD_4800: 4800 baud  BAUD_2400: 2400 baud  BAUD_1200: 1200 baud	BAUD_9600

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
ATR_DBITS Data length	If specified, the data length for a serial port is changed to the specified attribute values.	PORT	Read/Write	DBITS_5: 5 bits DBITS_6: 6 bits DBITS_7: 7 bits DBITS_8: 8 bits	DBITS_8
ATR_EOL End of line	If specified, the serial port is changed to terminate read when the specified attribute value. Refer to <a href="#">Appendix D, CHARACTER CODES</a> for a listing of valid attribute values.	PORT	Read/Write	Any ASCII character code	13 (carriage return)
ATR_FIELD Field	If specified, the amount of data read depends on the format specifier in the READ statement, or the default value of the data type being read. If not specified, the data is read until the terminator character (EOL) appears.	TP/CRT, PORT, FILE	Read only	Ignored	Read data until terminator character (EOL) appears
ATR_IA Interactively write	If specified, the contents of the buffer are output when each write operation to the buffer is complete. (Interactive)  If not specified, the contents of the buffer are output only when the buffer becomes full or when CR is specified. The size of the output buffer is 256 bytes. (Not interactive)	TP/CRT, PORT, FILE	Write only	Ignored	TP/CRT is interactive, PORT, FILE are not interactive

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
ATR_MODEM Modem line	Refer to the <a href="#">Modem line</a> in Section 7.2.1, Setting File and Port Attributes section that follows for information.				
ATR_PARITY Parity	The parity for a serial port can be changed to one of the valid attribute values.	PORT	Read/Write	PARITY_NONE: No parity  PARITY_ODD: Odd parity  PARITY_EVEN: Even parity	PARITY_NONE
ATR_PASSALL Passall	If specified, input is read without interpretation or transaction. Since the terminator character (EOL) will not terminate the read, the field attribute automatically assumes the field option.	TP/CRT, PORT	Read only	Ignored	Read only the displayable keys until enter key is pressed
ATR_PIPOVADV	Configures the behavior of the read when an overflow occurs. By default the behavior is to signal an end of file (EOF) when the overflow occurs.	PIPE	Read	The value must be between 0 and the total number of bytes in the pipe. The value will be rounded up to the nearest binary record.	The value parameter is either OVF_EOF (sets the default behavior) or the number of bytes to advance when an overflow occurs.
ATR_PIPWAIT	The read operation waits for data to arrive in the pipe.	PIPE	Read	WAIT_USED or WAIT_NOTUSED	The default is snapshot which means that the system returns an EOF when all the data in the pipe has been read.
ATR_READAHD Read Ahead Buffer	The attribute value is specified in units of 128 bytes, and allocates a read ahead buffer of the indicated size.	PORT	Read/Write	any positive INTEGER 1=128 bytes 2=256 bytes 0=disable bytes	1 (128 byte buffer)

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
ATR_REVERSE Reverse transfer	The bytes will be swapped.	PORT, FILE	Read/Write	Ignored	Not reverse transfer
ATR_SBITS Stop bits	This specifies the number of stop bits for the serial port.	PORT	Read/Write	SBITS_1:1 bit SBITS_15: 1.5 bits SBITS_2:2 bits	SBITS_1
ATR_TIMEOUT Timeout	If specified, an error will be returned by IO_STATUS if the read takes longer than the specified attribute value.	TP/CRT, PORT	Read only	Any INTEGER value (units are in msec)	0 (external)
ATR_UF Unformatted transfer	If specified, a binary transfer is performed. For read operations, the terminator character (EOL) will not terminate the read, and therefore automatically assumes the field option. If not specified, ASCII transfer is performed.	PORT, FILE	Read/Write	Ignored	ASCII transfer
ATR_XONOFF XON/XOFF	If specified, the XON/XOFF for a serial port is changed to the specified attribute value.	PORT	Read/Write	XF_NOT_USED: Not used XF_USED: Used	XF_USED

### Modem line

Valid device : PORT

Usage mode : Read/Write

Default value : MD\_NOT\_USED: DSR, DTR, and RTS not used

Valid attribute values :

- MD\_NOT\_USED: DSR, DTR, and RTS not used
- MD\_USE\_DSR: DSR used
- MD\_NOUSE\_DSR: DSR not used
- MD\_USE\_DTR: DTR used
- MD\_NOUSE\_DTR: DTR not used
- MD\_USE\_RTS: RTS used
- MD\_NOUSE\_RTS: RTS not used

- This attribute controls the operation of the modem line. The control is based on the following binary mask, where the flag bits are used to indicate what bit value you are changing.

RTS value	DSR value	DTR value	RTS flag	DSR flag	DTR flag
-----------	-----------	-----------	----------	----------	----------

- RTS (request to send) and DTR (data terminal ready) are both outputs.
- DSR (data set ready) is an input.

- Set the modem line attribute by doing the following.

- To indicate RTS is used (HIGH/ON): status = SET\_PORT\_ATR (port\_name, ATR\_MODEM, MD\_USE\_RTS)
- To indicate RTS is NOT used (LOW/OFF):status = SET\_PORT\_ATR (port\_name, ATR\_MODEM, MD\_NOUSE\_RTS)
- To indicate RTS is used (HIGH/ON) and DTR is not used (LOW/OFF):status = SET\_PORT\_ATR (port\_name, ATR\_MODEM, MD\_USE\_RTS or MD\_NOUSE\_DTR)
- The following examples demonstrate how to use the returned attribute value from the GET\_PORT\_ATR built-in.

```
status = GET_PORT_ATR (port, ATR_MODEM, atr_value)
```

- To determine if DTR is used:

```
IF ((atr_value AND MD_USE_DTR) = MD_USE_DTR) THEN
    write ('DTR is in use',cr)
ENDIF
```

- To determine if DTR is not used (LOW/OFF)

```
IF (atr_value AND MD_USE_DTR) = MD_NOUSE_DTR) THEN
    write ('DTR is not in use', cr)
ENDIF
```

For more information, refer to the [Section A.7.6, GET\\_PORT\\_ATR Built-In Function](#).

## 7.2.2 File String

The file string in an OPEN FILE statement specifies a data file name and type, or a communication port.

- The OPEN FILE statement associates the data file or port specified by the file string with the file variable. For example, OPEN FILE file\_var ('RO', 'data\_file.dt') associates the data file called 'data\_file.dt' with the file file\_var.
- If the file string is enclosed in single quotes, it is treated as a literal. Otherwise, it is treated as a STRING variable or constant identifier.
- When specifying a data file, you must include both a file name and a valid KAREL file type (any 1, 2, or 3 character file extension).
- The following STRING values can be used to associate file variables with serial communication ports on the KAREL controller. Defaults are:
  - 'P2:' – Debug console connector on the outside of the operator panel
  - 'P3:' – RS-232-C, JD17 connector on the Main CPU board (CRT/KB)
  - 'P4:' – RS-422, JD17 connector on the Main CPU board
  - 'KB:tp\_kb' – Input from numeric keypad on the teach pendant. TPDISPLAY or TPPROMPT are generally used, so OPEN FILE is not required.

- 'KB:cr\_kb' – Input from CRT/KB. INPUT or CRTPROMPT are generally used, so OPEN FILE is not required.
- 'WD>window\_name' – Writes to a window.
- 'WD>window\_name</keyboard\_name>', where keyboard\_name is either 'tpkb' or 'crkb' – Writes to the specified window. Inputs are from the TP keypad (tpkb) or the CRT keyboard (crkb). Inputs will be echoed in the specified window.

**See Also:** [Chapter 10, FILE SYSTEM](#) for a description of file names and file types.

### 7.2.3 Usage String

The usage string in an OPEN FILE statement indicates how the file is to be used.

- It is composed of one usage specifier.
- It applies only to the file specified by the OPEN FILE statement and has no effect on other FILES.
- It must be enclosed in single quotes if it is expressed as a literal.
- It can be expressed as a variable or a constant.

**Table 7.2.3** lists each usage specifier, its function, and the devices or ports for which it is intended.

- TP/CRT indicates teach pendant and CRT/KB.
- PORTS indicates serial ports.
- FILES indicates data files.
- PIPES indicates pipe devices.
- valid indicates a permissible use.
- no use indicates a permissible use that might have unpredictable side effects.

**Table 7.2.3 Usage Specifiers**

SPECIFIER	FUNCTION	TP/CRT	PORTS	FILES	PIPES
<b>RO</b>	<ul style="list-style-type: none"> <li>– Permits only read operations</li> <li>– Sets file position to beginning of file</li> <li>– File must already exist</li> </ul>	valid	valid	valid	valid
<b>RW</b>	<ul style="list-style-type: none"> <li>– Rewrites over existing data in a file, deleting existing data</li> <li>– Permits read and write operations</li> <li>– Sets file position to beginning of file</li> <li>– File will be created if it does not exist</li> </ul>	valid	valid	valid no use on FRx:	valid
<b>AP</b>	<ul style="list-style-type: none"> <li>– Appends to end of existing data</li> <li>– Permits read and write (First operation must be a write.)</li> <li>– Sets file position to end of file</li> <li>– File will be created if it does not exist</li> </ul>	no use	valid	valid -RAM disk* no use on FRx:	valid

SPECIFIER	FUNCTION	TP/CRT	PORTS	FILES	PIPES
<b>UD</b>	<ul style="list-style-type: none"> <li>- Updates from beginning of existing data. (Number of characters to be written must equal number of characters to be replaced.)</li> <li>- Overwrites the existing data with the new data</li> <li>- Permits read and write</li> <li>- Sets file position to beginning of existing file</li> </ul>	no use	valid	valid -RAM disk* no use on FRx :	no use

\* AP and UD specifiers can only be used with uncompressed files on the RAM disk. Refer to [Chapter 10, FILE SYSTEM](#), for more information on the RAM disk and Pipe devices.

[Figure 7.2.3](#) shows a program that includes examples of various file strings in OPEN FILE statements. The CONST and VAR sections are included to illustrate how file and port strings are declared.

```

PROGRAM open_luns
CONST
    part_file_c ='parts.dt' --data file STRING constant
    comm_port = 'P3:'           --port STRING constant
VAR
    file_var1 : FILE
    file_var2 : FILE
    file_var3 : FILE
    file_var4 : FILE
    file_var5 : FILE
    file_var12 : FILE
    temp_file : STRING[19]
        --a STRING size of 19 accommodates 4 character device names,
        --12 character file names, the period, and 2 character,
        --file types.
    port_var : STRING[3]
BEGIN
    --literal file name and type
    OPEN FILE file_var1 ('RO','log_file.dt')
    --constant specifying parts.dt
    OPEN FILE file_var2 ('RW', part_file_c)
    --variable specifying new_file_dt
    temp_file = 'RD:new_file.dt'
    OPEN FILE file_var3 ('AP', temp_file)
    --literal communication port
    OPEN FILE file_var4 ('RW', 'P2:')
    --constant specifying C0:
    OPEN FILE file_var5 ('RW', comm_port)
    --variable specifying C3:
    port_var = 'C3:'
    OPEN FILE file_var12 ('RW', port_var)
END open_luns

```

**Figure 7.2.3 File String Examples**

**See Also:** [Chapter 10, FILE SYSTEM](#), for more information on the available storage devices

[Chapter 16, INPUT/OUTPUT SYSTEM](#), for more information on the C0: and C3: ports

## 7.3 CLOSE FILE STATEMENT

---

The CLOSE FILE statement is used to break the association between a specified file variable and its data file or communication port. It accomplishes two objectives:

- Any buffered data is written to the file or port.
- The file variable is freed for another use.

[Figure 7.3](#) shows a program that includes an example of using the CLOSE FILE statement in a FOR loop, where several files are opened, read, and then closed. The same file variable is used for each file.

```
PROGRAM read_files
  VAR
    file_var      : FILE
    file_names   : ARRAY[10] OF STRING[15]
    loop_count   : INTEGER
    loop_file    : STRING[15]
  ROUTINE read_ops(file_spec:FILE) FROM util_prog
  --performs some read operations
  ROUTINE get_names(names:ARRAY OF STRING) FROM util_prog
  --gets file names and types
  BEGIN
    get_names(file_names)
    FOR loop_count = 1 TO 10 DO
      loop_file = file_names[loop_count]
      OPEN FILE file_var ('RO', loop_file)
      read_ops(file_var) --call routine for read operations
      CLOSE FILE file_var
    ENDFOR END read_files
```

**Figure 7.3 CLOSE FILE Example**

**See Also:** [Section A.3.12, CLOSE FILE Statement](#), [Section A.9.19, IO\\_STATUS Built-In Function](#), for a description of errors.

## 7.4 READ STATEMENT

---

The READ statement is used to read one or more specified data items from the indicated device. The data items are listed as part of the READ statement. The following rules apply to the READ statement:

- The OPEN FILE statement must be used to associate the file variable with the file opened in the statement before any read operations can be performed unless one of the predefined files is used (refer to [Table 7.1](#)).
- If the file variable is omitted from the READ statement, then TPDISPLAY is used as the default.
- Using the %CRTDEVICE directive will change the default to INPUT (CRT input window).
- Format specifiers can be used to control the amount of data that is read for each data item. The effect of format specifiers depends on the data type of the item being read and on whether the data is in text (ASCII) or binary (unformatted) form.
- When the READ statement is executed (for ASCII files), data is read beginning with the next nonblank input character and ending with the last character before the next blank, end of line, or end of file for all input types except STRING.

- With STRING values, the input field begins with the next character and continues to the end of the line or end of the file. If a STRING is read from the same line following a nonstring field, any separating blanks are included in the STRING.
- ARRAY variables must be read element by element; they cannot be read in unsubscripted form. Frequently, they are read using a READ statement in a FOR loop.
- PATH variables can be specified as follows in a READ statement, where *path\_name* is a PATH variable and *n* and *m* are PATH node indexes:
  - *path\_name* : specifies that the entire path, starting with a header and including all of the nodes and their associated data, is to be read. The header consists of the path length and the associated data description in effect when the PATH was written.
  - *path\_name [0]* : specifies that only the header is to be read. The path header consists of the path length and the associated data description in effect when the PATH was written. Nodes are deleted or created to make the path the correct length, and all new nodes are set uninitialized.
  - *path\_name [n]* : specifies that data is to be read into node[n] from the current file position. The value of *n* must be in the range from 0 to the length of the PATH.
  - *path\_name [n .. m]* : specifies that data is to be read into nodes *n* through *m*. The value of *n* must be in the range from 0 to the length of the PATH and can be less than, equal to, or greater than the value of *m*. The value of *m* must be in the range from 1 to the length of the PATH.

If an error occurs while reading node *n* (where *n* is greater than 0), it is handled as follows:

If *n* > original path length (prior to the read operation), the nodes from *n* to the new path length are set uninitialized.

If *n* <= original path length, the nodes from *n* to the original path length remain as they were prior to the read operation and any new nodes (greater than the original path length) are set uninitialized.

- If the associated data description that is read from the PATH does not agree with the current user associated data, the read operation is terminated and the path will remain as it was prior to the read operation. The IO\_STATUS built-in function will return an error if this occurs.
- PATH data must be read in binary (unformatted) form.

[Figure 7.4](#) shows several examples of the READ statement using a variety of file variables and data lists.

```

READ (next_part_no)      --uses default TPDISPLAY
OPEN FILE file_var ('RO','data_file.dt')
READ file_var (color, style, option)
READ host_line (color, style, option, CR)
FOR i = 1 TO array_size DO
    READ data (data_array[i])
ENDFOR

```

**Figure 7.4 READ Statement Examples**

If any errors occur during input, the variable being read and all subsequent variables up to CR in the data list are set uninitialized unless the file variable is open to a window device.

If reading from a window device, an error message is displayed indicating the bad data\_item and you are prompted to enter a replacement for the invalid data\_item and to reenter all subsequent items.

The built-in function IO\_STATUS can be used to determine the success or failure (and the reason for the failure) of a READ operation.

**See Also:** [Section A.18.1, READ Statement](#) , [Section A.9.19, IO\\_STATUS Built-In Function](#) for a list of I/O error messages, [Section A.3.54, %CRTDEVICE Translator Directive](#)

## 7.5 WRITE STATEMENT

---

The WRITE statement is used to write one or more specified data items to the indicated device. The data items are listed as part of the WRITE statement. The following rules apply to the WRITE statement:

- The OPEN FILE statement must be used to associate the file variable with the file opened in the statement before any write operations can be performed unless one of the predefined files is used (refer to [Table 7.1](#)).
- If the file variable is omitted from the WRITE statement, then TPDISPLAY is used as the default.
- Using the %CRTDEVICE directive will change the default to OUTPUT (CRT output window).
- Format specifiers can be used to control the format of data that is written for each data\_item. The effect of format specifiers depends on the data type of the item being written and on whether the data is in text (ASCII) or binary (unformatted) form.
- ARRAY variables must be written element by element; they cannot be written in unsubscripted form. Frequently, they are written using a WRITE statement in a FOR loop.
- PATH variables can be specified as follows in a WRITE statement, where *path\_name* is a PATH variable and *n* and *m* are PATH node indexes:
  - *path\_name* : specifies that the entire path is to be written, starting with a header that provides the path length and associated data table, and followed by all of the nodes, including their associated data.
  - *path\_name [0]* : specifies that only the header is to be written. The path header consists of the path length and a copy of the associated data table.
  - *path\_name [n]* : specifies that node[n] is to be written.
  - *path\_name [n .. m]* : specifies that nodes n through m are to be written. The value of n must be in the range from 0 to the length of the PATH and can be less than, equal to, or greater than the value of m. The value of m must be in the range from 1 to the length of the PATH.
- PATH data must be written in binary (unformatted) form.

[Figure 7.5](#) shows several examples of the WRITE statement using a variety of file variables and data lists.

```

WRITE TPPROMPT('Press T.P. key "GO" when ready')
  WRITE TP FUNC (' GO      RECD      QUIT      BACK1      FWD-1')
  WRITE log_file (part_no:5, good_count:5, bad_count:5, operator:3,
CR)
  WRITE ('This is line 1', CR, 'This is line 2', CR)
--uses default TPDISPLAY
FOR i = 1 TO array_size DO
  WRITE data (data_array[i])
ENDFOR

```

**Figure 7.5 WRITE Statement Examples**

See Also: [Section A.23.5, WRITE Statement](#), [Section A.9.19, IO\\_STATUS Built-In Function](#)

## 7.6 INPUT/OUTPUT BUFFER

---

An area of RAM, called a *buffer*, is used to hold up to 256 bytes of data that has not yet been transmitted during a read or write operation.

Buffers are used by the READ and WRITE statements as follows:

- During the execution of a READ statement, if more data was read from the file than required by the READ statement, the remaining data is kept in a buffer for subsequent read operations. For example, if you enter more data in a keyboard input line than is required to satisfy the READ statement the extra data is kept in a buffer.
- If a WRITE statement is executed to a non-interactive file and the last data item was not a CR, the data is left in a buffer until a subsequent WRITE either specifies a CR or the buffer is filled.
- The total data that can be processed in a single READ or WRITE statement is limited to 254 bytes.

## 7.7 FORMATTING TEXT (ASCII) INPUT/OUTPUT

This section explains the format specifiers used to read and write ASCII (formatted) text for each data type.

The following rules apply to formatting data types:

- For text files, data items in READ and WRITE statements can be of any of the simple data types (INTEGER, REAL, BOOLEAN, and STRING).
- Positional and VECTOR variables cannot be read from text files but can be used in WRITE statements.
- ARRAY variables cannot be read or written in unsubscripted form. The elements of an ARRAY are read or written in the format that corresponds to the data type of the ARRAY.
- PATH variables cannot be read or written.
- Some formats and data combinations are not read in the same manner as they were written or become invalid if read with the same format.

The amount of data that is read or written can be controlled using zero, one, or two format specifiers for each data item in a READ or WRITE statement. Each format specifier, represented as an INTEGER literal, is preceded by double colons (::).

[Table 7.7 \(a\)](#) summarizes the input format specifiers that can be used with the data items in a READ statement. The default format of each data type and the format specifiers that can affect each data type are explained in [Section 7.8.1, Formatting INTEGER Data Items](#), through [Section 7.7.6, Formatting Positional Data Items](#).

**Table 7.7 (a) Text (ASCII) Input Format Specifiers**

DATA TYPE	1ST FORMAT SPECIFIER	2ND FORMAT SPECIFIER
INTEGER	Total number of characters read	Number base in range 2 - 16
REAL	Total number of characters read	Ignored
BOOLEAN	Total number of characters read	Ignored
STRING	Total number of characters read	0 - unquoted STRING 2 - quoted STRING

[Table 7.7 \(b\)](#) summarizes the output format specifiers that can be used with the data items in a WRITE statement. The default format of each data type and the format specifiers that can affect each data type are explained in [Section 7.8.1, Formatting INTEGER Data Items](#).

**Table 7.7 (b) Text (ASCII) Output Format Specifiers**

DATA TYPE	1ST FORMAT SPECIFIER	2ND FORMAT SPECIFIER
INTEGER	Total number of characters written	Number base in range 2-16

DATA TYPE	1ST FORMAT SPECIFIER	2ND FORMAT SPECIFIER
REAL	Total number of characters written	Number of digits to the right of decimal point to be written If negative, uses scientific notation
BOOLEAN	Total number of characters written	0 - Left justified 1 - Right justified
STRING	Total number of characters written	0 - Left justified 1 - Right justified 2 - Left justified in quotes (leading blank) 3 - Right justified n quotes (leading blank)
VECTOR	Uses REAL format for each component	Uses REAL format for each component
POSITION	Uses REAL format for each component	Uses REAL format for each component
XYZWPR	Uses REAL format for each component	Uses REAL format for each component
XYZWPREXT	Uses REAL format for each component	Uses REAL format for each component
JOINTPOSn	Uses REAL format for each component	Uses REAL format for each component

## 7.7.1 Formatting INTEGER Data Items

INTEGER data items in a READ statement are processed as follows:

### Default:

Read as a decimal (base 10) INTEGER, starting with the next nonblank character on the input line and continuing until a blank or end of line is encountered. If the characters read do not form a valid INTEGER, the read operation fails.

### First Format Specifier:

Indicates the total number of characters to be read. The input field must be entirely on the current input line and can include leading, but not trailing, blanks.

### Second Format Specifier:

Indicates the number base used for the input and must be in the range of 2 (binary) to 16 (hexadecimal).

For bases over 10, the letters A, B, C, D, E, and F are used as input for the digits with values 10, 11, 12, 13, 14, and 15, respectively. Lowercase letters are accepted.

Table 7.7.1 (a) lists examples of INTEGER input data items and their format specifiers. The input data and the resulting value of the INTEGER data items are included in the table. (The symbol [eol] indicates end of line.)

**Table 7.7.1 (a) Examples of INTEGER Input Data Items**

DATA ITEM	INPUT DATA	RESULT
int_var	-2[eol]	int_var = -2
int_var	20 30 ...	int_var = 20
int_var::3	10000	int_var = 100
int_var::5::2	10101 (base 2 input)	int_var = 21 (base 10 value)
int_var	1.00	format error (invalid INTEGER)
int_var::5	100[eol]	format error (too few digits)

INTEGER data items in a WRITE statement are formatted as follows:

**Default:**

Written as a decimal (base 10) INTEGER using the required number of digits and one leading blank. A minus sign precedes the digits if the INTEGER is a negative value.

**First Format Specifier:**

Indicates the total number of characters to be written, including blanks and minus sign. If the format specifier is larger than required for the data, leading blanks are added. If it is smaller than required, the field is extended as required.

The specifier must be in the range of 1 to 127, otherwise the length is used.

**Second Format Specifier:**

Indicates the number base used for the output and must be in the range of 2 (binary) to 16 (hexadecimal).

If a number base other than 10 (decimal) is specified, the number of characters specified in the first format specifier (minus one for the leading blank) is written, with leading zeros added if needed.

For bases over 10, the letters A, B, C, D, E, and F are used as input for the digits with values 10, 11, 12, 13, 14, and 15, respectively.

**Table 7.7.1 (b)** lists examples of INTEGER output data items and their format specifiers. The output values of the INTEGER data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**Table 7.7.1 (b) Examples of INTEGER Output Data Items**

DATA ITEM	OUTPUT	COMMENT
123	" 123"	Leading blank
-5	" -5 "	Leading blank
123::6	"123"	Right justified (leading blanks)
-123::2	" -123"	Expanded as required
1024::0::16	" 400"	Hexadecimal output
5::6::2	" 00101"	Binary output (leading zeros)

DATA ITEM	OUTPUT	COMMENT
-1::9::16	" FFFFFFFF"	Hexadecimal output

## 7.7.2 Formatting REAL Data Items

REAL data items in a READ statement are processed as follows:

**Default:**

Read starting with the next nonblank character on the input line and continuing until a blank or end of line is encountered.

Data can be supplied with or without a fractional part. The E used for scientific notation can be in upper or lower case. If the characters do not form a valid REAL, the read operation fails.

**First Format Specifier:**

Indicates the total number of characters to be read. The input field must be entirely on the current input line and can include leading, but not trailing, blanks.

**Second Format Specifier:**

Ignored for REAL data items.

**Table 7.7.2 (a)** lists examples of REAL input data items and their format specifiers. The input data and the resulting value of the REAL data items are included in the table. The symbol [eol] indicates end of line and X indicates extraneous data on the input line.

**Table 7.7.2 (a) Examples of REAL Input Data Items**

DATA ITEM	INPUT DATA	RESULT
real_var	1[eol]	1.0
real_var	1.000[eol]	1.0
real_var	2.5 XX	2.50
real_var	1E5 XX	100000.0
real_var::7	2.5 XX	format error (trailing blank)
real_var	1E	format error (no exponent)
real_var::4	1E 2	format error (embedded blank)

REAL data items in a WRITE statement are formatted as follows:

**Default:** Written in scientific notation in the following form:

```
(blank) (msign) (d) . (d) (d) (d) (d) E (esign) (d) (d)
```

where:

```
(blank) is a single blank
(msign) is a minus sign, if required
(d) is a digit
(esign) is a plus or minus sign
```

**First Format Specifier:**

Indicates the total number of characters to be written, including all the digits, blanks, signs, and a decimal point. If the format specifier is larger than required for the data, leading blanks are added. If it is smaller than required, the field is extended as required.

In the case of scientific notation, character length should be greater than (8 + 2nd format specifier) to write the data completely.

The specifier must be in the range of 1 to 127, otherwise 13 is used.

#### **Second Format Specifier:**

Indicates the number of digits to be output to the right of the decimal point, whether or not scientific notation is to be used.

The absolute value of the second format specifier indicates the number of digits to be output to the right of the decimal point.

If the format specifier is positive, the data is displayed in fixed format (that is, without an exponent). The maximum value is 12. If it is negative, scientific notation is used. The minimum value is -6.

**Table 7.7.2 (b)** lists examples of REAL output data items and their format specifiers. The output values of the REAL data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**Table 7.7.2 (b) Examples of REAL Output Data Items**

DATA ITEM	OUTPUT	COMMENT
123.0	" 1.23000E+02"	Scientific notation (default format)
123.456789	" 1.23457E+02"	Rounded to 5 digits in fractional part
.00123	" 1.23000E-03"	Negative exponent
-1.00	" -1.00000E+00"	Negative value
-123.456::9	" -1.234560E+02"	Field expanded
123.456::12	" 1.234560E+02"	Leading blank added
123.456::9::2	" 123.46"	Right justified and rounded
123.::12::3	" 1.230E+02"	Scientific notation

## **7.7.3 Formatting BOOLEAN Data Items**

BOOLEAN data items in a READ statement are formatted as follows:

#### **Default:**

Read starting with the next nonblank character on the input line and continuing until a blank or end of line is encountered.

Valid input values for TRUE include TRUE, TRU, TR, T, and ON. Valid input values for FALSE include FALSE, FALS, FAL, FA, F, OFF, and OF. If the characters read do not form a valid BOOLEAN, the read operation fails.

#### **First Format Specifier:**

Indicates the total number of characters to be read. The input field must be entirely on the current input line and can include leading, but not trailing, blanks.

#### **Second Format Specifier:**

Ignored for BOOLEAN data items.

**Table 7.7.3 (a)** lists examples of BOOLEAN input data items and their format specifiers. The input data and the resulting value of the BOOLEAN data items are included in the table. (The symbol [eol] indicates end of line and X indicates extraneous data on the input line.)

**Table 7.7.3 (a) Examples of BOOLEAN Input Data Items**

DATA ITEM	INPUT DATA	RESULT
bool_var	FALSE[eol]	FALSE
bool_var	FAL 3...	FALSE
bool_var	T[eol]	TRUE
bool_var::1	FXX	FALSE (only reads "F")
bool_var	O[eol]	format error (ambiguous)
bool_var	1.2[eol]	format error (not BOOLEAN)
bool_var::3	F [eol]	format error (trailing blanks)
bool_var::6	TRUE[eol]	format error (not enough data)

BOOLEAN data items in a WRITE statement are formatted as follows:

**Default:**

Written as either "TRUE" or "FALSE". (Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.)

**First Format Specifier:**

Indicates the total number of characters to be written, including blanks (a leading blank is always included). If the format specifier is larger than required for the data, trailing blanks are added. If it is smaller than required, the field is truncated on the right.

The specifier must be in the range of 1 to 127, otherwise the length is used.

**Second Format Specifier:**

Indicates whether the data is left or right justified. If the format specifier is equal to 0, the output word is left justified in the output field with one leading blank, and trailing blanks as required. If it is equal to 1, the output word is right justified in the output field, with leading blanks as required.

**Table 7.7.3 (b)** lists examples of BOOLEAN output data items and their format specifiers. The output values of the BOOLEAN data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**Table 7.7.3 (b) Examples of BOOLEAN Output Data Items**

DATA ITEM	OUTPUT	COMMENT
FALSE	" FALSE"	Default includes a leading blank
TRUE	" TRUE"	TRUE is shorter than FALSE
FALSE::8	"FALSE "	Left justified (default)
FALSE::8::1	" FALSE"	Right justified
TRUE::2	" T"	Truncated

## 7.7.4 Formatting STRING Data Items

---

STRING data items in a READ statement are formatted as follows:

**Default:**

Read starting at the current position and continuing to the end of the line. If the length of the data obtained is longer than the declared length of the STRING, the data is truncated on the right. If it is shorter, the current length of the STRING is set to the actual length.

**First Format Specifier:**

Indicates the total field length of the input data. If the field length is longer than the declared length of the STRING, the input data is truncated on the right. If it is shorter, the current length of the STRING is set to the specified field length.

**Second Format Specifier:**

Indicates whether or not the input STRING is enclosed in single quotes. If the format specifier is equal to 0, the input is not enclosed in quotes.

If it is equal to 2, the input must be enclosed in quotes. The input is scanned for the next nonblank character. If the character is not a quote, the STRING is not valid and the read operation fails.

If the character is a quote, the remaining characters are scanned until another quote or the end of line is found. If another quote is not found, the STRING is not valid and the read operation fails.

If both quotes are found, all of the characters between them are read into the STRING variable, unless the declared length of the STRING is too short, in which case the data is truncated on the right.

Table 7.7.4 (a) lists examples of STRING input data items and their format specifiers, where str\_var has been declared as a STRING[5]. The input data and the resulting value of the STRING data items are included in the table. The symbol [eol] indicates end of line and X indicates extraneous data on the input line.

**Table 7.7.4 (a) Examples of STRING Input Data Items**

DATA ITEM	INPUT DATA	RESULT
str_var	"ABC [eol]"	"ABC"
str_var	"ABCDEFG [eol]"	"ABCDE" (FG is read but the STRING is truncated to 5 characters)
str_var	" 'ABC' XX"	" 'AB" (blanks and quote are read as data)
str_var::0::2	" 'ABC' XX"	" 'ABC'" (read ends with second quote)

STRING data items in a WRITE statement are formatted as follows:

**Default:**

Content of the STRING is written with no trailing or leading blanks or quotes.

**First Format Specifier:**

Indicates the total number of characters to be written, including blanks. If the format specifier is larger than required for the data, the data is left justified and trailing blanks are added. If the format specifier is smaller than required, the STRING is truncated on the right.

The specifier should be in the range of 1 to 254, otherwise the string length is used.

### Second Format Specifier:

Indicates whether the output is to be left or right justified and whether the STRING is to be enclosed in quotes using the following values:

- 0 left justified, no quotes
- 1 right justified, no quotes
- 2 left justified, quotes
- 3 right justified, quotes

Quoted STRING values, even if left justified, are preceded by a blank. Unquoted STRING values are not automatically preceded by a blank.

**Table 7.7.4 (b)** lists examples of STRING output data items and their format specifiers. The output values of the STRING data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**Table 7.7.4 (b) Examples of STRING Output Data Items**

DATA ITEM	OUTPUT	COMMENT
'ABC'	"ABC"	No leading blanks
'ABC'::2	"AB"	Truncated on right
'ABC'::8	"ABC "	Left justified
'ABC'::8::0	"ABC "	Same as previous
'ABC'::8::1	" ABC"	Right justified
'ABC'::8::2	" 'ABC' "	Note leading blank
'ABC'::8::3	" 'ABC'"	Right justified
'ABC'::4::2	"'A'"	Truncated

Format specifiers for STRING data items can cause the truncation of the original STRING values or the addition of trailing blanks when the values are read again.

If STRING values must be successively written and read, the following guidelines will help you ensure that STRING values of varying lengths can be read back identically:

- The variable into which the STRING is being read must have a declared length at least as long as the actual STRING that is being read, or truncation will occur.
- Some provision must be made to separate STRING values from preceding variables on the same data line. One possibility is to write a ' ' (blank) between a STRING and the variable that precedes it.
- If format specifiers are not used in the read operation, write STRING values at the ends of their respective data lines (that is, followed in the output list by a CR) because STRING variables without format specifiers are read until the end of the line is reached.
- The most general way to write string values to a file and read them back is to use the format ::0::2 for both the read and write.

## 7.7.5 Formatting VECTOR Data Items

VECTOR data items cannot be read from text (ASCII) files. However, you can read three REAL values and assign them to the elements of a VECTOR variable. VECTOR data items in a WRITE statement are formatted as three REAL values on the same line.

[Table 7.7.5](#) lists examples of VECTOR output data items and their format specifiers, where vect.x = 1.0, vect.y= 2.0, vect.z = 3.0. The output values of the VECTOR data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**See Also:** [Section 7.7.2, Formatting REAL Data Items](#) , for information on the default output format and format specifiers used with REAL data items

**Table 7.7.5 Examples of VECTOR Output Data Items**

DATA ITEM	OUTPUT
vect	" 1. 2. 3."
vect::6::2	" 1.00 2.00 3.00"
vect::12::-3	" 1.000E+00 2.000E+00 3.000E+00"

## 7.7.6 Formatting Positional Data Items

Positional data items cannot be read from text (ASCII) files. However, you can read six REAL values and a STRING value and assign them to the elements of an XYZWPR variable or use the POS built-in function to compose a POSITION. The CNV\_STR\_CONF built-in can be used to convert a STRING to a CONFIG data type.

POSITION and XYZWPR data items in a WRITE statement are formatted in three lines of output. The first line contains the location (x,y,z) component of the POSITION, the second line contains the orientation (w,p,r), and the third line contains the configuration string.

The location and orientation components are formatted as six REAL values. The default format for the REAL values in a POSITION is the default format for REAL(s). Refer to [Section 7.7.2, Formatting REAL Data Items](#) .

The configuration string is not terminated with a CR, meaning you can follow it with other data on the same line.

[Table 7.7.6](#) lists examples of POSITION output data items and their format specifiers, where p = POS(2.0,-4.0,8.0,0.0,90.0,0.0,config\_var) . The output values of the POSITION data items are also included in the table. Double quotes are used in the table as delimiters to show leading blanks; however, double quotes are not written by KAREL programs.

**Table 7.7.6 Examples of POSITION Output Data Items (p = POS(2.0,-4.0,8.0,0.0,90.0,0.0,config\_var))**

DATA ITEM	OUTPUT
p	" 2. -4. 8." " 0. 9. 0." "N, 127, , -1"

DATA ITEM	OUTPUT
p:7::2	" 2.00-4.00 8.00" " 0.0090.00 0.00" "N, 127, , -1"

JOINTPOS data items in a WRITE statement are formatted similarly to POSITION types with three values on one line.

**See Also:** [Section 7.7.2, Formatting REAL Data Items](#), for information on format specifiers used with REAL data items, [Section A.16.12, POS Built-In Function](#)

## 7.8 FORMATTING BINARY INPUT/OUTPUT

This section explains the format specifier used in READ and WRITE statements to read and write binary (unformatted) data for each data item. Binary input/output operations are sometimes referred to as unformatted, as opposed to text (ASCII) input/output operations that are referred to as formatted.

The built-in SET\_FILE\_ATR with the ATR\_UF attribute is used to designate a file variable for binary operations. If not specified, ASCII text operations will be used.

Data items in READ and WRITE statements can be any of the following data types for binary files:

- INTEGER
- REAL
- BOOLEAN
- STRING
- VECTOR
- POSITION
- XYZWPR
- XYZWPREXT
- JOINTPOS

Vision and array variables cannot be read or written in unsubscripted form. The elements of an ARRAY are read or written in the format that corresponds to the data type of the ARRAY.

Entire PATH variables can be read or written, or you can specify that only node[0] (containing the PATH header), a specific node, or a range of nodes be read or written. Format specifiers have no effect on PATH data. PATH data can be read or written only to a file and not to a serial port, CRT/KB, or teach pendant.

Binary I/O is preferred to text I/O when creating files that are to be read only by KAREL programs for the following reasons:

- Positional, VECTOR, and PATH variables cannot be read directly from text input.
- Some formats and data combinations are not read in the same manner as they were written in text files or they become invalid if read with the same format.
- Binary data is generally more compact, reducing both the file size and the I/O time.
- There is some inevitable loss of precision when converting from REAL data to its ASCII representation and back.

Generally, no format specifiers need to be used with binary I/O. If this rule is followed, all input data can be read exactly as it was before it was written.

However, if large numbers of INTEGER values are to be written and their values are known to be small, writing these with format specifiers reduces both storage space and I/O time.

For example, INTEGER values in the range of -128 to +127 require only one byte of storage space, and INTEGER values in the range of -32768 to +32767 require two bytes of storage space. Writing INTEGER values in these ranges with a first format specifier of 1 and 2, respectively, results in reduced storage space and I/O time requirements, with no loss of significant digits.

**Table 7.8** summarizes input and output format specifiers that can be used with the data items in READ and WRITE statements. The default format of each data type is also included. [Section 7.7.1, Formatting INTEGER Data Items](#) through [Section 7.7.6, Formatting Positional Data Items](#) explain the effects of format specifiers on each data type in more detail.

**See Also:** [Section A.19.13, SET\\_FILE\\_ATR Built-In Procedure](#)

**Table 7.8 Binary Input/Output Format Specifiers**

DATA TYPE	DEFAULT	1ST FORMAT SPECIFIER	2ND FORMAT SPECIFIER
INTEGER	Four bytes read or written	Specified number of least significant bytes read or written, starting with most significant (1-4)	Ignored
REAL	Four bytes read or written	Ignored	Ignored
BOOLEAN	Four bytes read or written	Specified number of least significant bytes read or written, starting with most significant (1-4)	Ignored
STRING	Current length of string (1 byte), followed by data bytes	Number of bytes read or written	Ignored
VECTOR	Three 4-byte REAL numbers read or written	Ignored	Ignored
POSITION	60 bytes read or written	Ignored	Ignored
XYZWPR	36 bytes read or written	Ignored	Ignored
XYZWPREXT	48 bytes read or written	Ignored	Ignored
JOINTPOSn	8 + n*4 bytes read or written	Ignored	Ignored
PATH	Depends on size of structure	Ignored	Ignored

## 7.8.1 Formatting INTEGER Data Items

INTEGER data items in a READ or WRITE statement are formatted as follows:

### Default:

Four bytes of data are read or written starting with the most significant byte.

### First Format Specifier:

Indicates the number of least significant bytes of the INTEGER to read or write, with the most significant of these read or written first. The sign of the most significant byte read is extended to unread bytes. The format specifier must be in the range from 1 to 4.

For example, if an INTEGER is written with a format specifier of 2, bytes 3 and 4 (where byte 1 is the most significant byte) will be written. There is no check for loss of significant bytes when INTEGER values are formatted in binary I/O operations.

**NOTE**

Formatting of INTEGER values can result in undetected loss of high order digits.

**Second Format Specifier:**

Ignored for INTEGER data items.

## 7.8.2 Formatting REAL Data Items

REAL data items in a READ or WRITE statement are formatted as follows:

**Default:**

Four bytes of data are read or written starting with the most significant byte.

**First Format Specifier:**

Ignored for REAL data items.

**Second Format Specifier:**

Ignored for REAL data items.

## 7.8.3 Formatting BOOLEAN Data Items

BOOLEAN data items in a READ or WRITE statement are formatted as follows:

**Default:**

Four bytes of data are read or written. In a read operation, the remainder of the word, which is never used, is set to 0.

**First Format Specifier:**

Indicates the number of least significant bytes of the BOOLEAN to read or write, the most significant of these first. The format specifier must be in the range from 1 to 4. Since BOOLEAN values are always 0 or 1, it is always safe to use a field width of 1.

**Second Format Specifier:**

Ignored for BOOLEAN data items.

## 7.8.4 Formatting STRING Data Items

STRING data items in a READ or WRITE statement are formatted as follows:

**Default:**

The current length of the STRING (not the declared length) is read or written as a single byte, followed by the content of the STRING. STRING values written without format specifiers have their lengths as part of the output, while STRING values written with format specifiers do not. Likewise, if a STRING is read without a format

specifier, the length is expected in the data, while if a STRING is read with a format specifier, the length is not expected.

This means that, if you write and then read STRING data, you must make sure your use of format specifiers is consistent.

**First Format Specifier:**

Indicates the number of bytes to be read or written.

**Second Format Specifier:**

Ignored for STRING data items.

In a read operation, if the first format specifier is greater than the declared length of the STRING, the data is truncated on the right. If it is less than the declared length of the STRING, the current length of the STRING is set to the number of bytes read.

In a write operation, if the first format specifier indicates a shorter field than the current length of the STRING, the STRING data is truncated on the right. If it is longer than the current length of the STRING, the output is padded on the right with blanks.

Writing STRING values with format specifiers can cause truncation of the original STRING values or padding blanks on the end of the STRING values when reread.

---

## 7.8.5 Formatting VECTOR Data Items

---

VECTOR data items in a READ or WRITE statement are formatted as follows:

**Default:**

Data is read or written as three 4-byte binary REAL numbers.

**First Format Specifier:**

Ignored for VECTOR data items.

**Second Format Specifier:**

Ignored for VECTOR data items.

---

## 7.8.6 Formatting POSITION Data Items

---

POSITION data items in a READ or WRITE statement are formatted as follows:

**Default:**

Read or written in the internal format of the controller, which is 60bytes long.

---

## 7.8.7 Formatting XYZWPR Data Items

---

XYZWPR data items in a READ or WRITE statement are formatted as follows:

**Default:**

Read or written in the internal format of the controller, which is 36 bytes long.

## 7.8.8 Formatting XYZWPREXT Data Items

XYZWPREXT data items in a READ or WRITE statement are formatted as follows:

**Default:**

Read or written in the internal format of the controller, which is 48 bytes long.

## 7.8.9 Formatting JOINTPOS Data Items

JOINTPOS data items in a READ or WRITE statement are formatted as follows:

**Default:**

Read or written in the internal format of the controller, which is 8bytes plus 4 bytes for each axis.

## 7.9 USER INTERFACE TIPS

Input and output to the teach pendant or CRT/KB is accomplished by executing READ and WRITE statements within a KAREL program. If the **USER** menu is not the currently selected menu, the input will remain pending until the **USER** menu is selected. The output will be written to the saved windows that will be displayed when the **USER** menu is selected. You can have up to eight saved windows.

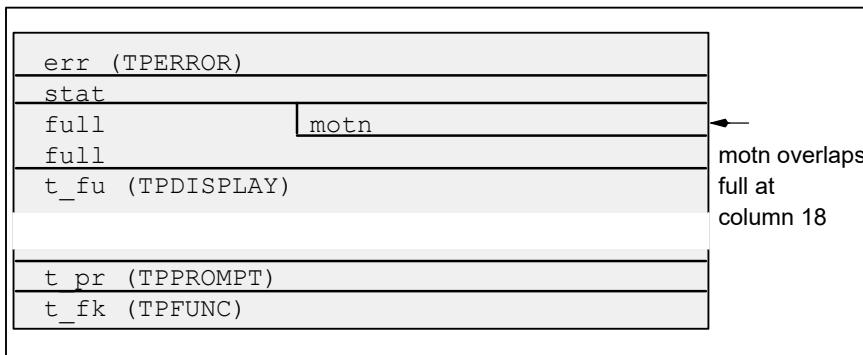
### 7.9.1 USER Menu on the Teach Pendant

The screen that is activated when the **USER** menu is selected from the teach pendant is named **t\_sc**. The windows listed in [Table 7.9.1](#) are defined for **t\_sc**.

**Table 7.9.1 Defined Windows for t\_sc**

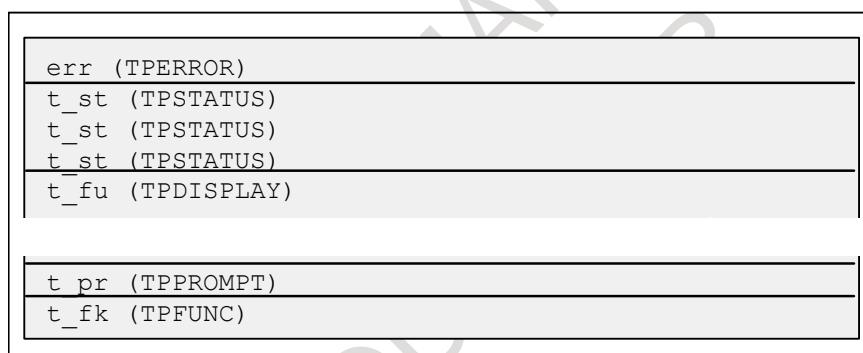
Window Name	Lines	Predefined FILE Name	Scrolled	Rows
t_fu	10	TPDISPLAY	yes	5-14
t_pr	1	TPPROMPT	no	15
t_st	3	TPSTATUS	no	2-4
t_fk	1	TPFUNC	no	16
err	1	TPERROR	no	1
stat	1		no	2
full	2		no	3-4
motn	1		no	3

By default, the **USER** menu will attach the **err**, **stat**, **full**, **motn**, **t\_fu**, **t\_pr**, and **t\_fk** windows to the **t\_sc** screen. See [Figure 7.9.1 \(a\)](#).

**Figure 7.9.1 (a) t\_sc Screen**

The following system variables affect the teach pendant **USER** menu:

- **\$TP\_DEFPROG: STRING** - Identifies the teach pendant default program. This is automatically set when a program is selected from the teach pendant **SELECT** menu.
- **\$TP\_INUSER: BOOLEAN** - Set to TRUE when the **USER** menu is selected from the teach pendant.
- **\$TP\_LCKUSER: BOOLEAN** - Locks the teach pendant in the **USER** menu while **\$TP\_DEFPROG** is running and **\$TP\_LCKUSER** is TRUE.
- **\$TP\_USESTAT: BOOLEAN** - Causes the user status window **t\_st** (TPSTATUS) to be attached to the user screen while **\$TP\_USESTAT** is TRUE. While **t\_st** is attached, the **stat**, **motn**, and **full** windows will be detached. See [Figure 7.9.1 \(b\)](#).

**Figure 7.9.1 (b) t\_sc Screen with \$TP\_USESTAT = TRUE**

## 7.9.2 USER Menu on the CRT/KB

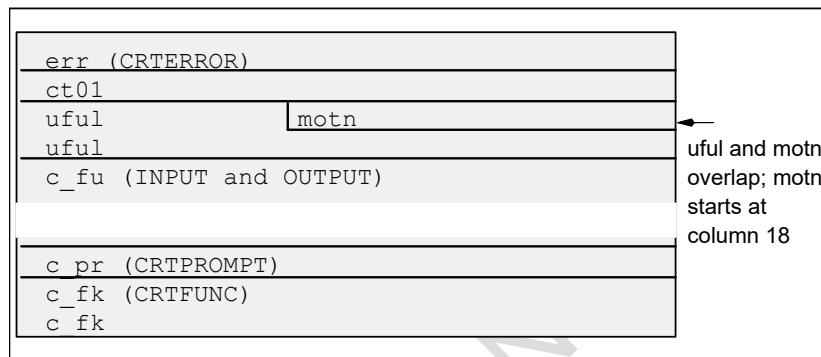
The screen that is activated when the **USER** menu is selected from the CRT is named **c\_sc**. The windows listed in [Table 7.9.2](#) are defined for **c\_sc**.

**Table 7.9.2 Defined Windows for c\_sc**

Window Name	Lines Predefined FILE Name	Predefined FILE Name	Scrolled	Rows
c_fu	17	INPUT and OUTPUT	yes	5-21
c_pr	1	CRTPROMPT	no	22
c_st	3	CRTSTATUS	no	2-4
c_fk	2	CRTFUNC	no	23-24

Window Name	Lines Predefined FILE Name	Predefined FILE Name	Scrolled	Rows
err	1	CRTERROR	no	1
ct01	1		no	2
uful	2		no	3-4
motn	1		no	3

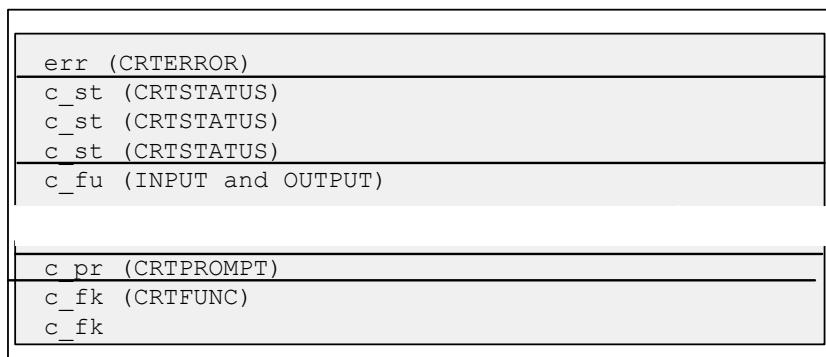
By default, the **USER** menu will attach the **err**, **ct01**, **uful**, **motn**, **c\_fu**, **c\_fk**, and **uftn** windows to the **c\_sc** screen. The **c\_fk** window will label the function keys and show **FCTN** and **MENUS** for F9 and F10. See [Figure 7.9.2 \(a\)](#).



**Figure 7.9.2 (a) c\_sc Screen**

The following system variables affect the CRT USER menu:

- **\$CRT\_DEFPROG: STRING** - This variable identifies the CRT default program. This is automatically set when a program is selected from the CRT SELECT menu.
- **\$CRT\_INUSER: BOOLEAN** - This variable is set to TRUE when the **USER** menu is selected from the CRT.
- **\$CRT\_LCKUSER: BOOLEAN** - This variable locks the CRT in the **USER** menu while **\$CRT\_DEFPROG** is running and **\$CRT\_LCKUSER** is TRUE.
- **\$CRT\_USERSTAT: BOOLEAN** - This variable causes the user status window **c\_st** (CRTSTATUS) to be attached to the user screen while **\$CRT\_USERSTAT** is TRUE. While **c\_st** is attached, the **ct01**, **motn**, and **uful** windows will be detached. See [Figure 7.9.2 \(b\)](#).



**Figure 7.9.2 (b) c\_sc Screen with \$CRT\_USERSTAT = TRUE**

# 8 POSITION DATA

In robotic applications, single segment motion is the movement of the tool center point (TCP) from an initial position to a desired destination position. The KAREL system represents positional data in terms of location (x, y, z), orientation (w, p, r), and configuration. The location and orientation are defined relative to a Cartesian coordinate system (user frame), making them independent of the robot joint angles. Configuration represents the unique set of joint angles at a particular location and orientation.

## NOTE

The KAREL system provides a way to create and manipulate position data but it does not support motion instructions. All motion must be initiated from a teach pendant program. Instructions and built-ins are available for setting KAREL position data into teach pendant program.

## 8.1 POSITIONAL DATA

The KAREL language uses the POSITION, XYZWPR, XYZWPREXT, JOINTPOS, and PATH data types to represent positional data. The POSITION data type is composed of the following:

- Three REAL values representing an x, y, z location expressed in millimeters
- Three REAL values representing a w, p, r orientation expressed in degrees
- One CONFIG Data Type, consisting of 4 booleans and 3 integers, which represent the configuration in terms of joint placement and turn number. Before you specify the config data type, make sure it is valid for the robot being used. Valid joint placement values include:
  - R or L (shoulder right or left)
  - U or D (elbow up or down)
  - N or F (wrist no-flip or flip)
  - T or B (config front or back)

A turn number is the number of complete turns a multiple turn joint makes beyond the required rotation to reach a position. [Table 8.1](#) lists the valid turn number definitions.

**Table 8.1 Turn Number Definitions**

Turn Number	Rotation (degrees)
-8	-2700 to -3059
-7	-2340 to -2699
-6	-1980 to -2339
-5	-1620 to -1979
-4	-1260 to -1619
-3	-900 to -1259
-2	-540 to -899
-1	-180 to -539
0	-179 to 179

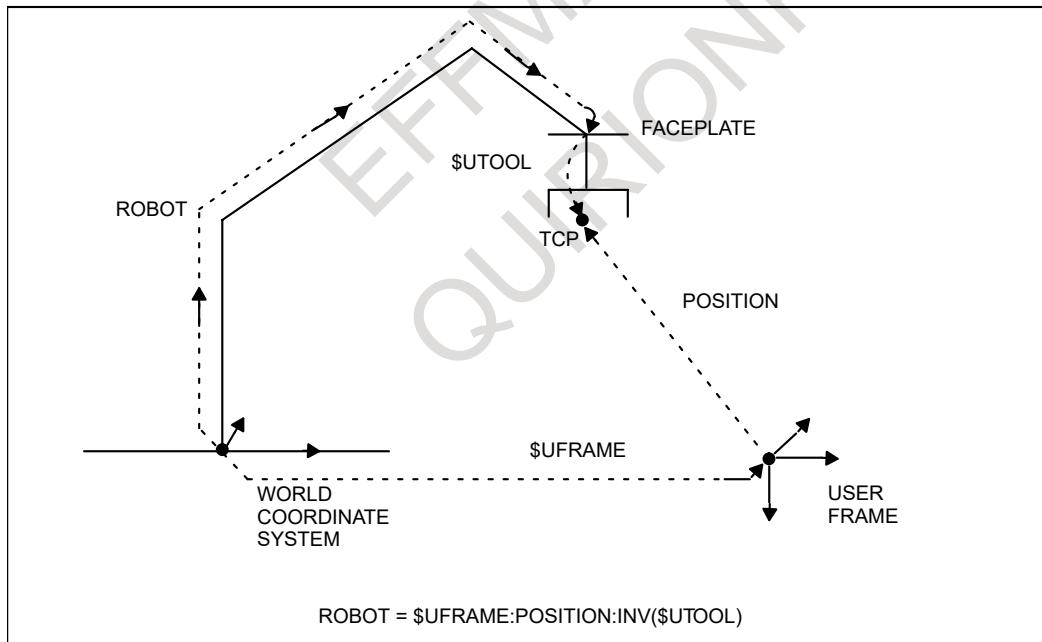
Turn Number	Rotation (degrees)
1	180 to 539
2	540 to 899
3	900 to 1259
4	1260 to 1619
5	1620 to 1979
6	1980 to 2339
7	2340 to 2699

The PATH data type consists of a varying-length list of elements called *path nodes*.

See Also: Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*) for configuration information on each supported robot model. [Section A.16.15, POSITION Data Type](#), [Section A.24.6, XYZWPR Data Type](#), [Section A.24.7, XYZWPREXT Data Type](#), [Section A.10.2, JOINTPOS Data Type](#), [Section A.16.1, PATH Data Type](#)

## 8.2 FRAMES OF REFERENCE

The KAREL system defines the location and orientation of positional data relative to a user-defined frame of reference, called user frame, as shown in [Figure 8.2](#).



**Figure 8.2 Referencing Positions in KAREL**

Three frames of reference exist:

- WORLD - predefined
- UFRAME - determined by the user
- UTOOL - defined by the tool

Using kinematic equations, the controller computes its positional information based on the known world frame and the data stored in the system variables `$UFRAME` (for user frame) and `$UTOOL` (for tool frame).

## 8.2.1 World Frame

---

The world frame is predefined for each robot. It is used as the default frame of reference. The location of world frame differs for each robot model.

## 8.2.2 User Frame (UFRAME)

---

The programmer defines user frame relative to the world frame by assigning a value to the system variable `$UFRAME`.

 **WARNING**

Be sure `$UFRAME` is set to the same value whether you are teaching positional data or running a program with that data, or damage to the tool could occur.

The location of UFRAME represents distances along the x-axis, y-axis, and z-axis of the world coordinate system; the orientation represents rotations around those axes.

By default, the system assigns a (0,0,0) location value and a (0,0,0) orientation value to `$UFRAME`, meaning the user frame is identical to that of the world coordinate system. All positions are recorded relative to UFRAME.

## 8.2.3 Tool Definition (UTOOL)

---

The *tool center point (TCP)* is the origin of the UTOOL frame of reference. The programmer defines the position of the TCP relative to the faceplate of the robot by assigning a value to the system variable `$UTOOL`. By default, the system assigns a (0,0,0) location and a (0,0,0) orientation to `$UTOOL`, meaning `$UTOOL` is identical to the faceplate coordinate system. The positive z-axis of UTOOL defines the *approach vector* of the tool.

 **WARNING**

Be sure `$UTOOL` correctly defines the position of the TCP for the tool you are using, or damage to the tool could occur.

The faceplate coordinate system has its origin at the center of the faceplate surface. Its orientation is defined with the plane of the x-axis and y-axis on the faceplate and the positive z-axis pointing straight out from the faceplate.

## 8.2.4 Using Frames in the Teach Pendant Editor (TP)

The system variable `$USEUFRAME` defines whether the current value of `$MNUFRAMENUM[group_no]` will be assigned to the position's user frame when it is being recorded or touched up.

- When `$USEUFRAME = FALSE`, the initial recording of positions and the touching up of positions is done with the user frame number equal to 0, regardless of the value of `$MNUFRAMENUM[group_no]`.
- When `$USEUFRAME = TRUE`, the initial recording of positions is done with the position's user frame equal to the user frame defined by `$MNUFRAMENUM[group_no]`. The touching up of positions must also be done with the position's user frame equal to the user frame defined by `$MNUFRAMENUM[group_no]`.

When a position is recorded in the teach pendant editor, the value of the position's tool frame will always equal the value of `$MNUTOOLNUM[group_no]` at the time the position was recorded. When a teach pendant program is executed, you must make sure that the user frame and the tool frame of the position equal the values of `$MNUFRAMENUM[group_no]` and `$MNUTOOLNUM[group_no]`; otherwise, an error will occur. Set the values of `$MNUFRAMENUM[1]` and `$MNUTOOLNUM[1]` using the `UFRAME_NUM=n` and `UTOOL_NUM=n` instructions in the teach pendant editor before you record the position to guarantee that the user and tool frame numbers match during program execution.

## 8.3 JOG COORDINATE SYSTEMS

The KAREL system provides five different jog coordinate systems:

### JOINT:

A joint coordinate system in which individual robot axes move. The motion is joint interpolated.

### WORLD:

A Cartesian coordinate system in which the TCP moves parallel to, or rotates around, the x, y, and z-axes of the predefined WORLD frame. The motion is linearly interpolated.

### TOOLFRAME:

A Cartesian coordinate system in which the TCP moves parallel to, or rotates around, the x, y, and z-axes of the currently selected tool frame. The motion is linearly interpolated. The tool frame is normally selected using the **SETUP Frames** menu. To jog using `$GROUP[group_no].$utool`, set `$MNUTOOLNUM[group_no] = 30`.

### JOGFRAME:

A Cartesian coordinate system in which the TCP moves parallel to, or rotates around, the x, y, and z-axes of the coordinate system defined by the `$JOG_GROUP[group_no].$jogframe` system variable. The motion is linearly interpolated.

### USER FRAME:

A Cartesian coordinate system in which the TCP moves parallel to, or rotates around, the x, y, and z-axes of the currently selected user frame. The motion is linearly interpolated. The user frame is normally selected using the **SETUP Frames** menu. To jog using `$GROUP[group_no].$uframe`, set `$MNUFRAMENUM[group_no] = 62`.

The robot can be jogged in any one of these jog coordinate systems to reach a destination position. Once that position is reached, however, the positional data is recorded with reference to the user frame as discussed in [Section 8.2, FRAMES OF REFERENCE](#).

See Also: Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*) for step-by-step explanations of how to jog and define frames.

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# 9 TEACHING KAREL VARIABLES

---

KAREL variables consist of the following:

- Positions
- Paths
- Variables

Positions and paths used in a program must be *taught*, or defined, before you can run the program or run production.

Variables used in a program can be defined outside of a program, if they have not been defined within the program.

This section contains information about teaching KAREL positions, paths, and variables.

## 9.1 KAREL Positions

---

Before you can run a KAREL program or run production using a KAREL program, you must *teach* all KAREL positions within the program. You can use two methods to teach a KAREL position:

- **Jog** the robot to the position and **record** it.
- **Set** the value of each positional component.

You can set KAREL positions from the **DATA KAREL Posns** screen.

**Table 9.1 DATA KAREL Posns Screen Items**

ITEM	DESCRIPTION
Position name	This item is the name of the position.
Motion group	This item indicates that the position is in a particular motion group.
Position status	This item indicates whether the position has been taught.

This section contains a procedure for teaching KAREL positions.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*) for information on jogging the robot, and for information on the components of a position. Use [Section 9.1.1, Teaching KAREL Positions](#) to teach KAREL positions.

Use [Section 9.1.1, Teaching KAREL Positions](#) to teach KAREL positions.

### 9.1.1 Teaching KAREL Positions

---

#### Before you begin

- All personnel and unnecessary equipment are out of the workcell.
- The KAREL program that contains the positions you are teaching has been loaded into controller memory.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)*.

## Procedure

1. Select the KAREL program for which you want to modify KAREL variables:
  - a) Press **SELECT**.
  - b) Press **F1, [TYPE]**.
  - c) Select **KAREL Progs.**
  - d) Move the cursor to the KAREL program or variable file you want and press **ENTER**.
2. Press **DATA**.
3. Press **F1, [TYPE]**.
4. Select **KAREL Posns**. See the following screen for an example of the *top level KAREL Posns* screen. It lists both position and path variables.

```
DATA KAREL Posns
1 POS1          G1  Not Recorded
2 POS2          G2  Not Recorded
3 WPR1          G1  Not Recorded
4 POS_ARR1     G1  [5] of Position
5 WPR_ARR2     G1  [4,5] of XYZWPR
6 VEC1          Vector
7 PTH1          2 nodes
8 PTH2          0 nodes
9 PTH3          0 nodes
10 PTH4         3 nodes
```

### NOTE

**G1** indicates that the position is in motion group 1.

**@**, when displayed, indicates that the robot is currently near the position.

### NOTE

If KAREL positions are not displayed, make sure the **\$KAREL\_ENB** system variable is set to 1.

### NOTE

You can now teach a position by recording it or by setting position components. After you have taught a position you can test it by moving the robot to it.

5. To teach a position by recording it:
  - a) On the top level **KAREL Posns** screen, move the cursor to the position you want to teach.
  - b) Continuously press the **DEADMAN** switch and turn the teach pendant **ON/OFF** switch to **ON**.
  - c) Jog the robot to the position you want to record.
  - d) Press and hold the **SHIFT** key and press **F3, RECORD**. The status of the position changes from **Not Recorded** to **Recorded**.

**6.** To teach a position by setting position components:

- On the top level **KAREL Posns** screen, move the cursor to the position you want to teach.
- Press **ENTER**.

You will see a screen similar to the following.

```
DATA KAREL Posns
POS1 IN GROUP[1]
1 C Not Recorded
2 X Not Recorded
3 Y Not Recorded
4 Z Not Recorded
5 W Not Recorded
6 P Not Recorded
7 R Not Recorded
```

c) Select the position component you want to set:

- **To set configuration, C**, move the cursor to **C** and press **ENTER**. Select the configuration settings you want and set them. Press **PREV** when you are finished setting configuration.
- **To set X, Y, Z, W, P, or R**, move the cursor to the component and use the numeric keys to type the value. Press **ENTER**.

d) Press **PREV** to return to the top level **KAREL Posns** screen.

**7.** If you want to move the robot to a taught position:

- On the top level **KAREL Posns** screen, move the cursor to the position to which you want to move the robot.

**NOTE**

The position you want to move the robot to must have been previously taught.

- Press and hold in the **DEADMAN** switch and turn the teach pendant **ON/OFF** switch to ON.

**⚠ WARNING**

In the next step, the robot will move. Make sure that personnel and unnecessary equipment are out of the workcell, otherwise, personnel could be injured and equipment damaged.

c) Press the key for the type of move you want the robot to perform:

- **For a linear move**, press and hold in the **SHIFT** key and press **F4, MOVE\_LN**. When the robot starts moving, you can release **F4** but you must continue pressing the **SHIFT** key.
- **For a joint move**, press and hold in the **SHIFT** key and press **F5, MOVE\_JT**. When the robot starts moving, you can release **F5** but you must continue pressing the **SHIFT** key.

**8.** To save the positions to a file:

- From any of the **KAREL Posn** screens, press **FCTN**.
- Select **SAVE**. All the variables in the selected KAREL program will be saved to the file, **PROGRAM.VR**, on the default device.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*) for information on setting the device.

## 9.2 KAREL Paths

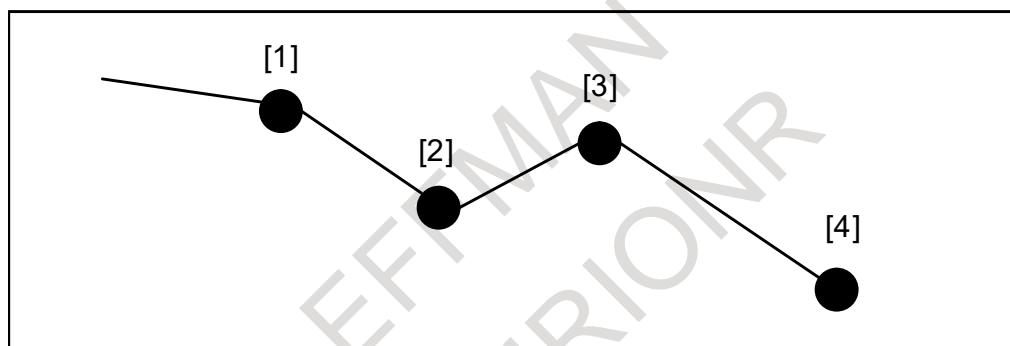
Before you can run a KAREL program or run production using a KAREL program, you must *teach* all KAREL paths within the program. You can use two methods to teach a KAREL path:

- **Jog** the robot to the position of each path node and **record** it.
- **Set** the value of each position component of each path node.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*) for information on jogging the robot, and for information on the components of a position. Use [Section 9.1.1, Teaching KAREL Positions](#) to teach KAREL positions.

### KAREL Path

A *path* can be thought of as a list of positions that has one name for the entire list. A *path node* is an individual position in the list. See [Figure 9.2](#).



**Figure 9.2 Path with Four Path Nodes**

You refer to an individual path node by specifying its number in the list. The number follows the path variable name and is enclosed in brackets ([ ]). `WELD_PATH[4]`, for example, is the fourth node in the path called `WELD_PATH`. You can have a standard path node, user-defined path node, and a user-defined header.

When you use a path to represent a series of positions, you do not need to know in advance how many positions the path includes. When you teach the path, you can use as many positions as you need. Each position you teach is added to the list of nodes for a path.

In a path you can do the following:

- **Append** or add new path nodes to the end of the path
- **Insert** new path nodes any place in the path except at the end
- **Delete** or remove path nodes from the path

You can also teach associated data that accompanies each path. In addition, if you have specified path header information, you can teach the path header information when teaching other path information.

### Standard KAREL Path Node

A standard KAREL path node contains a list of the standard node types. Standard path node types are

- NODE\_POS
- GROUP\_ASSOC
- COMMON\_ASSOC

**NODE\_POS** is the position of the path node. It contains positional components you can teach by recording or by setting values.

**GROUP\_ASSOC** and **COMMON\_ASSOC** are *path associated data*. Path associated data defines information specific to the path node.

### User-Defined KAREL Path Node

A user-defined path node contains any information you defined in the KAREL program.

### User-Defined Header

A user-defined *header* contains more information associated with a path. The header is defined in the KAREL program as part of the path, if desired. You can set header information outside of a KAREL program, if necessary.

Use [Section 9.2.1, Teaching KAREL Paths](#) to teach a KAREL path.

## 9.2.1 Teaching KAREL Paths

---

### Before you begin

- All personnel and unnecessary equipment are out of the workcell.
- The KAREL program that contains the paths you are teaching has been loaded into controller memory.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).

### Procedure

1. Select the KAREL program for which you want to modify KAREL variables:
  - a) Press **SELECT**.
  - b) Move the cursor to the KAREL program you want and press **ENTER**.
2. Press **DATA**.
3. Press **F1, [TYPE]**.
4. Select **KAREL Posns**. See the following screen for an example of the top level **KAREL Posns** screen. It lists both position and path variables.

```
DATA KAREL Posns
1 POS1          G1  Not Recorded
2 POS2          G2  Not Recorded
3 WPR1          G1  Not Recorded
4 POS_ARR1     G1  [5] of Position
5 WPR_ARR2     G1  [4,5] of XYZWPR
6 VEC1          Vector
7 PTH1          2 nodes
```

8 PTH2	0 nodes
9 PTH3	0 nodes
10 PTH4	3 nodes

**NOTE**

**G1** indicates that the position is in motion group 1.

@, when displayed, indicates that the robot is currently near the position.

**NOTE**

If KAREL paths are not displayed, make sure the **\$KAREL\_ENB** system variable is set to 1.

5. Move the cursor to the path you want to teach and press **ENTER**. You will see a screen similar to the following.

```
DATA KAREL Posns
PTH2
1  [1]          *
```

A list of path nodes is displayed. If this is a new path and no nodes exist one node will be created for you automatically. If you do not record or add any nodes during your path editing session and you press **PREV**, this node will be deleted.

**NOTE**

\* indicates the path node has not been recorded. R, when displayed, indicates the path node has been recorded.

@, when displayed, indicates the path node has been recorded and the robot is near that node.

**NOTE**

You can now teach each path node by recording it or by setting position components. After you have taught a path node you can test it by moving the robot to it.

6. To teach a path node by recording it:
  - a) Move the cursor to the path node you want to teach.
  - b) Press and hold in the **DEADMAN** switch and turn the teach pendant **ON/OFF** switch to **ON**.
  - c) Jog the robot to the position you want to record.
  - d) Press and hold the **SHIFT** key and press **F3, RECORD**.

**NOTE**

Recording a node moves the cursor to the next node in the list automatically. If you record the last node, a new node is appended to the end of the list automatically. If you do not record this node and you press **PREV**, this new node will be deleted.

- e) Repeat [Step 6 \(a\)](#) through [Step 6 \(d\)](#) for each path node you want to record.
- 7.** To teach a path node by setting position components:
- Move the cursor to the path node you want to teach.
  - Press **ENTER**. See the following screen for an example of a standard path node.

```
DATA KAREL Posns
PTH2[1]
1 NODE_POS           G1 Not Recorded
2 GROUP_DATA          Group Assoc
3 COMMON_DATA         Common Assoc
```

- c) Select **NODE\_POS** and press **ENTER**. You will see a screen similar to the following.

```
DATA KAREL Posns
PTH2[1].NODE_POS IN GROUP[1]
1 C                 Not Recorded
2 X                 Not Recorded
3 Y                 Not Recorded
4 Z                 Not Recorded
5 W                 Not Recorded
6 P                 Not Recorded
7 R                 Not Recorded
```

- d) Select the position component you want to set:
- **To set configuration, C**, move the cursor to **C** and press **ENTER**. Select the configuration settings you want and set them. Press **PREV** when you are finished setting configuration.
  - **To set X, Y, Z, W, P, or R**, move the cursor to the component and use the numeric keys to type the value. Press **ENTER**.
- e) Press **PREV** until the following screen is displayed (list of node information).

```
DATA KAREL Posns
PTH2[1]
1 NODE_POS           G1 Recorded
2 GROUP_DATA          Group Assoc
3 COMMON_DATA         Common Assoc
```

- f) **To display information for the next path node**, press **F2, NXNODE**.
- g) **To display information for the previous path node**, press **NEXT,>**, and press **F2, PRNODE**.
- h) **To display information for a specific path node**, at the list of path nodes, press **NEXT, >**, press **F3, IXNODE**, and then enter the number of the path node you want.
- i) Repeat [Step 7 \(a\)](#) through [Step 7 \(d\)](#) for each node in the path.

- 8.** To teach path associated data:
- Move the cursor to the path node you want to teach.
  - Press **ENTER**. See the following screen for an example of a standard path node.

```
DATA KAREL Posns
PTH2[1]
1 NODE_POS           G1 Recorded
2 GROUP_DATA          Group Assoc
```

3 COMMON_DATA	Common Assoc
---------------	--------------

- c) Select the associated data you want to teach (GROUP\_DATA, COMMON\_DATA, or other user-defined data) and press **ENTER**. You will see a screen similar to the following.

```
DATA KAREL Posns
PTH2[1].GROUP_DATA
1 SEGRELSPEED          0
2 SEGTYPE
3 SEGORIENTYPE
4 SEGBREAK             FALSE
```

- d) Select an item and set it to the value you want.  
e) When you are finished with this group of data, press **PREV**.  
f) Repeat [Step 8 \(a\)](#) through [Step 8 \(d\)](#) for each group of associated data you want to teach.

## 9. To teach path header information:

- a) Press **PREV** until the top level **KAREL Posns** screen is displayed. See the following screen for an example.

### NOTE

You can teach path header information only if you have defined it in your KAREL program. If **HEADER** appears as the function key label for **F2** when you select a path, path header information has been defined and you can teach it.

```
DATA KAREL Posns
1 POS1      G1  Not Recorded
2 POS2      G2  Not Recorded
3 WPR1      G1  Not Recorded
4 POS_ARR1  G1  [5] of Position
5 WPR_ARR2 G1  [4,5] of XYZWPR
6 VEC1      Vector
7 PTH1      2 nodes
8 PTH2      4 nodes
9 PTH3      0 nodes
10 PTH4     3 nodes
```

### NOTE

**G1** indicates that the position is in motion group 1.

@, when displayed, indicates that the robot is currently near the position.

- b) Move the cursor to the path for which you want to modify header information.  
c) Press **F2, HEADER**. You will see a screen similar to the following.

```
DATA KAREL Posns
PTH2.PATHHEADER
1 INT1      *uninit*
2 INT2      *uninit*
3 S1        STRUCT1_T
```

- d) Select the item you want to teach and enter the necessary information.
  - e) When you are finished editing the path header information, press **PREV** to return to the top level **KAREL Posns** screen.
- 10.** Press **PREV** until the top level **KAREL Posns** screen is displayed. See the following screen for an example.

```
DATA KAREL Posns
1 POS1           G1  Not Recorded
2 POS2           G2  Not Recorded
3 WPR1           G1  Not Recorded
4 POS_ARR1      G1  [5] of Position
5 WPR_ARR2      G1  [4,5] of XYZWPR
6 VEC1           Vector
7 PTH1           2 nodes
8 PTH2           4 nodes
9 PTH3           0 nodes
10 PTH4          3 nodes
```

#### NOTE

**G1** indicates that the position is in motion group 1.

@, when displayed, indicates that the robot is currently near the position.

- 11.** If you want to modify a path (append, delete, or insert path nodes), select the path you want to modify and press **ENTER**. You will see a screen similar to the following.

```
DATA KAREL Posns
PTH2
1 [1]           R
2 [2]           R
3 [3]           R
4 [4]           R
```

#### NOTE

**R** indicates that the path node has been recorded.

\* indicates that the path node has not been recorded.

@ indicates that the path node has been recorded and the robot is near that node.

- 12. To append a path node to the end of the list:**

- a) Move the cursor to any path node.
- b) Press **NEXT, >**.
- c) Press **F2, APPEND**. A node is appended to the end of the list. The cursor is moved to the newly appended node.
- d) Teach the path node by recording or setting it.

- 13. To delete a path node:**

**NOTE**

You cannot delete a path node if it is the only one in the list.

**14.** Move the cursor to the path node you want to delete.

**15.** Press **NEXT, >**.

**16.** Press **F3, DELETE**.

**17. To insert a path node between two path nodes:**

- Decide where you want to insert the path node. Move the cursor to select the path node following that path node. For example, if you want to insert a path node between nodes 5 and 6, place the cursor on path node 6.
- Press **NEXT, >**.
- Press **F4, INSERT**. A node is inserted into the list. The cursor is moved to the newly inserted node.
- Teach the path node by recording or setting it.

**18. If you want to move the robot to a taught path node:**

- On the top level **KAREL Posns** screen, move the cursor to the path you want to use and press **ENTER** to display the list of path nodes. See the following screen for an example.

```
DATA KAREL Posns
PTH2
1 [1] R
2 [2] R
3 [3] R
4 [4] R
```

**NOTE**

The path node you want to move the robot to must have been previously taught.

- Press and hold in the **DEADMAN** switch and turn the teach pendant **ON/OFF** switch to **ON**.
- Set the speed to a low value for safety.

**⚠ WARNING**

In the next step, the robot will move. Make sure that personnel and unnecessary equipment are out of the workcell, otherwise, personnel could be injured and equipment damaged.

**19.** Press the key for the type of move you want the robot to perform:

- For a linear move, press and hold in the **SHIFT** key and press **F4, MOVE\_LN**. When the robot starts moving, you can release **F4** but must continue pressing **SHIFT**.
- For a joint move, press and hold in the **SHIFT** key and press **F5, MOVE\_JT**. When the robot starts moving, you can release **F5** but must continue pressing **SHIFT**.

**20. To save the path to a file:**

- a) From any of the **KAREL Posn** screens, press **FCTN**.
- b) Select **SAVE**. All the variables in the selected KAREL program will be saved to the file, **PROGRAM.VR**, on the default device. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (**B-83284EN**) for information on setting the device.

## 9.3 KAREL Variables

---

KAREL variables are created and defined within a KAREL program. You can set variables used in KAREL programs outside of the program, if necessary.

Refer to [Table 9.3](#) for information on the items on the **KAREL Vars** screen. Use [Section 9.3.1, Modifying KAREL Variables](#) to set KAREL variables outside of a program.

**Table 9.3 DATA KAREL Vars Screen Items**

ITEM	DESCRIPTION
Variable name	This item is a listing of the KAREL variables in the selected program.
Variable type	This item is a listing of the data type for each variable. To set the variable, move the cursor to the variable you want to change, press <b>ENTER</b> , and type the new value.

### 9.3.1 Modifying KAREL Variables

---

#### Before you begin

- The KAREL program that contains the variables you are modifying has been loaded into controller memory.

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (**B-83284EN**).

#### Procedure

1. Select the KAREL program whose variables you want to modify:
  - a) Press **SELECT**.
  - b) Press **F1, [TYPE]**.
  - c) Select **KAREL Progs**.
  - d) Move the cursor to the KAREL program you want and press **ENTER**.
2. Press **DATA**.
3. Press **F1, [TYPE]**.
4. Select **KAREL Vars**. You will see a screen similar to the following.

```
DATA KAREL Vars
1 I *uninit*
```

```
2 I_ARR1          [10] of Integer
3 R              *uninit*
4 R_ARR2          [3,4] of Real
5 BOOL            *uninit*
6 STR             *uninit*
7 S1              STRUCT1_T
8 S2              STRUCT2_T
9 S1_ARR1         [3] of STRUCT1_T
10 S2_ARR3        [3,4,5] of STRUCT2_T
```

**NOTE**

If KAREL variables are not displayed, make sure the **\$KAREL\_ENB** system variable is set to 1.

5. Move the cursor to the variable you want to set, press **ENTER**, and type the necessary information.
6. If the variable is an array, a list of array elements is displayed or press **PREV** to return to the top level KAREL variables screen. Move the cursor to the element or field you want to set and press **ENTER**. If the variable is a structure, a list of fields is displayed. Type the necessary information.
7. To save the variables to a file:
  - a) Press **MENU**.
  - b) Select **FILE**.
  - c) Press **F1, [TYPE]**.
  - d) Select **File**.
  - e) Press **F5, [UTIL]**.
  - f) Select **Set Device**.
  - g) Move the cursor to the device you want and press **ENTER**.
  - h) From any of the **KAREL Posn** screens, press **FCTN**.
  - i) Select **SAVE**. All the variables in the selected KAREL program will be saved to the file, **PROGRAM.VR**, on the default device.

# 10 FILE SYSTEM

---

The file system provides a means of storing data on CMOS RAM, FROM, or external storage devices. The data is grouped into units, with each unit representing a file. For example, a file can contain the following:

- Source code statements for a KAREL program
- A sequence of KCL commands for a command procedure
- Variable data for a program

Files are identified by file specifications that include the following:

- The name of the device on which the file is stored
- The name of the file
- The type of data included in the file

The KAREL system includes five types of storage devices where files can be stored:

- RAM Disk
- FROM Disk
- IBM PC
- Memory Card
- USB Memory Stick Device

**RAM Disk** is a portion of SRAM (formerly CMOS RAM) or DRAM memory that functions as a separate storage device. Any file can be stored on the RAM Disk. RAM Disk files should be copied to disks for permanent storage.

**FROM Disk** is a portion of FROM memory that functions as a separate storage device. Any file can be stored on the F-ROM disk. However, the hardware supports a limited number of read and write cycles. Therefore, if a file needs to store dynamically changing data, the RAM disk should be used instead.

**IBM PC** or compatible computers can be used to store files off-line. You can use the FTP server, or the PC SHARE feature of the Internet Protocol Connection and Customization option for the PC to store files on an external storage device. The files on these storage devices are accessible in the following ways:

- Through the FILE menu on the teach pendant and CRT/KB
- Through KAREL programs

**Memory Card** refers to the ATA Flash File storage. The memory card interface is located on the MAIN CPU.

For more information on storage devices and memory, refer to [Section 10.2, STORAGE DEVICE ACCESS](#).

**USB Memory Stick Device** supports a USB 1.1 interface. The USB Organization specifies standards for USB 1.1 and 2.0. Most memory stick devices conform to the USB 2.0 specification for operation and electrical standards. USB 2.0 devices as defined by the USB Specification must be backward compatible with USB 1.1 devices.

However, FANUC America Corporation does not support any security or encryption features on USB memory sticks. The controller supports most widely-available USB Flash memory sticks from 32MB up to 1GB in size. The USB interface is located on the controller operator panel.

## 10.1 FILE SPECIFICATION

---

File specifications identify files. The specification indicates:

- The name of the device on which the file is stored, refer to [Section 10.1.1, Device Name](#) .
- The name of the file, refer to [Section 10.1.2, File Name](#) .
- The type of data the file contains, refer to [Section 10.1.3, File Type](#) .

The general form of a file specification is:

```
device_name:file_name.file_type
```

## 10.1.1 Device Name

A device name consists of at least two characters that indicate the device on which a file is stored. Files can be stored on RAM disk, F-ROM disk, disk drive units, off-line on a PC, Memory Card, or PATH Composite Device. The device name always ends with a colon (:). The following is a list of valid storage devices.

### RD: (RAM Disk)

The RD: device name refers to files stored on the RAM disk of the controller. RD: is used as the default device name. The RAM disk is capable of storing program and other backups and any files. As standard, it is placed on DRAM, and all files are erased when the power is interrupted. It can be placed on SRAM with appropriate settings, in which case, it can retain information with its backup battery when power is interrupted.

### FR: (F-ROM Disk)

The FR: device name refers to files stored on the F-ROM disk of the controller. The F-ROM disk is capable of storing program and other backups and any files. It can retain information when power is interrupted, with no backup battery.

### MC: (Memory Card Device)

The Memory Card device can be a Flash ATA memory card or SRAM memory card. It is possible to use a Compact Flash card by attaching a PCMCIA adapter to it. The memory card can be formatted and used as an MS-DOS file system. It can be read from and written to on the controller and an IBM PC equipped with the proper hardware and software. If the memory card is used as an MS-DOS file system, it should be formatted only on the controller. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)* for information on formatting the memory card on the controller. The memory card slot is on the main board.

### UD1: (USB Memory Stick Device)

UD1: is USB memory mounted on the USB port on the operator panel. The USB memory stick can be formatted and used as an MS-DOS file system. It can be read from and written to on the controller and an IBM PC equipped with the proper hardware and software. If the USB memory stick is used as an MS-DOS file system, it should be formatted only on the controller. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)* for information on formatting the USB memory stick on the controller.

### UT1: (USB Memory Stick Device)

UT1: is USB memory mounted on the USB port on the iPendant. Option software A05B-2500-J957 (USB port on iPendant) is required to use UT1:

### MD: (Memory Device)

The memory device is capable of handling data on the memory of a controller, such as robot programs and KAREL programs, as files. The memory device treats the controller's program memory as if it were a file device. You can access all teach pendant programs, KAREL programs, KAREL variables, system variables, and error logs that are loaded in the controller. See [Section 10.5, MEMORY DEVICE](#) for further details.

#### **MDB: (Memory Device Backup)**

The memory device backup device (MDB:) allows the user to copy the same files as provided by the Backup function on the **FILE** menu. This allows you to back up the controller remotely.

#### **CONS: (Console Device)**

The console device provides access to the console log text files CONSLOG.LS and CONSTAIL.LS. It is used for diagnostic and debug purposes and not as a storage device.

#### **MF: (Memory File Device)**

The MF: disk is a device in which an F-ROM disk and a RAM disk are synthesized together. The file list on the MF disk displays the files on both the F-ROM disk and the RAM disk, so that the files on both disks can be read. When a backup is to be made to the MF disk, a confirmation message appears asking which of FR: and RD: to use for storage. The MF: device name refers to files stored on both the RAM and F-ROM disks. Since a file cannot be on both disks at the same time, there will be no duplicate file names.

#### **PATH: (Composite Device)**

The PATH: device is a read-only device that searches the F-ROM disk (FD:), memory card (MC:), and floppy disk (FLPY:) in that order, for a specified file. The PATH: device eliminates the user's need to know on which storage device the specified file exists.

#### **PIP: (File Pipe Device)**

The PIP: device provides a way to write data from one application and, at the same time, read it from another application. The PIP: device also allows the last set of data written to be retained for analysis. The PIP: device allows you to access any number of pipe files. This access is to files that are in the controller's memory. This means that the access to these files is very efficient. The size of the files and number of files are limited by available controller memory. This means that the best use of a file pipe is to buffer data or temporarily hold it.

#### **FTP (C1: to C8:)**

FTP devices write and read files to and from an FTP server, such as a PC connected via Ethernet. It is displayed only if FTP client settings have been made on the host communication screen.

---

## **10.1.2 File Name**

A file name is an identifier that you choose to represent the contents of a file.

The following rules apply to file names:

- File names are limited to 36 characters.
- File names can include letters, digits, and underscores.

- File names cannot include these characters: . : \* ; \ / " '
- Spaces are not allowed in the file name.
- Other special characters can be used with caution.
- Subdirectories can be used. These are also called a *path*. These begin and end with the \ character. The rules for file names also apply to paths. Below is an example of a file name with a device and a path:

MC : \mypath\myfile.txt

## 10.1.3 File Type

A file type consists of two or three characters that indicate what type of data a file contains. A file type always begins with a period (.). **Table 10.1.3** is an alphabetical list of each available file type and its function.

**Table 10.1.3 File Type Descriptions**

File Type	Description
.BMP	<b>Bit map files</b> contain bit map images used in robot vision systems.
.CF	<b>KCL command files</b> are ASCII files that contain a sequence of KCL commands for a command procedure.
.CH	<b>Condition handler files</b> are used as part of the condition monitor feature.
.DF	<b>Default files</b> are binary files that contain the default motion instructions for each teach pendant programming.
.DG	<b>Diagnostic files</b> are ASCII files that provide status or diagnostic information about various functions of the controller.
.DT	<b>KAREL data files</b> are ASCII or binary files that can contain any data that is needed by the user.
.IO	<b>Binary files</b> that contain I/O configuration data - generated when an I/O screen is displayed and the data is saved.
.KL	<b>KAREL source code files</b> are ASCII files that contain the KAREL language statements for a KAREL program.
.LS	<b>KAREL listing files</b> are ASCII files that contain the listing of a KAREL language program and line number for each KAREL statement.
.MN	<b>Mnemonic program files</b> are supported in previous versions of KAREL.
.ML	<b>Part model files</b> contain part model information used in robot vision systems.
.PC	<b>KAREL p-code files</b> are binary files that contain the p-code produced by the translator upon translation of a .KL file.
.SV	<b>System files</b> are binary files that contain data for tool and user frames (SYSFRAME.SV), system variables (SYSVARS.SV), mastering (SYSMAST.SV), servo parameters (SYSSERVO.SV), and macros (SYSMACRO.SV).
.TP	<b>Teach pendant program files</b> are binary files that contain instructions for teach pendant programs.
.TX	<b>Text files</b> are ASCII files that can contain system-defined text or user-defined text.
.VR	<b>Program variable files</b> are binary files that contain variable data for a KAREL program.

File Type	Description
.VA	<b>ASCII variable files</b> contain the listing of a variable file with variable names, types, and contents.
.LS	<b>Listing files</b> are teach pendant programs, error logs, and description histories in ASCII format.

## 10.2 STORAGE DEVICE ACCESS

---

The KAREL system can access only those storage devices that have been formatted and mounted. These procedures are performed when the devices are first installed on the KAREL system.

The following rules apply when accessing storage devices:

- Formatting a device
  - Deletes any existing data on the device. For example, if you format RD2:, you will also reformat any data existing on RD: thru RD7:.
  - Records a directory on the device
  - Records other data required by the KAREL system
  - Assigns a volume name to the device

For more information on formatting a device, refer to the [Section A.6.6, FORMAT\\_DEV Built-In Procedure](#) in [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#), or the [Section C.26, FORMAT command](#) in [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#)

### 10.2.1 Storage Device Access Overview

---

The following kinds of storage devices can be used to store programs and files:

- Memory Card (MC:) (not available on the R-30iiB Mate Plus Controller)
- Flash File Storage disk (FR:)
- RAM Disk (RD:)
- Ethernet Device (optional)
- Memory Device (MD:)
- Memory Device Binary (MDB:)
- Filtered Memory Device (FMD:)
- USB Memory Stick Device on the controller (UD1:)
- USB Memory Stick Device on the iPendant (UT1:)

#### NOTE

You can perform a backup to the UT1: device but you can not use UT1: to perform a full load or an Auto Update.

This section describes how to set up storage devices for use. Depending on the storage device, this can include

- Setting up a port on the controller
- Connecting the device to the controller
- Formatting a device

**Memory Card (MC:)**

The R-30iB Plus Controller includes a memory card interface. Memory cards are available in various sizes. Compact Flash PC cards are also supported if used with a suitable compact adapter. The memory card requires a memory card interface which is standard on Main CPU inside the controller. The controller also supports loading software from memory cards.

**NOTE**

The memory card interface is not available on the R-30iB Mate Plus Controller.

**⚠ WARNING**

Lethal voltage is present in the controller WHENEVER IT IS CONNECTED to a power source. Be extremely careful to avoid electrical shock. HIGH VOLTAGE IS PRESENT at the input side whenever the controller is connected to a power source. Turning the disconnect or circuit breaker to the OFF position removes power from the output side of the device only.

**⚠ WARNING**

The memory card interface is located on the Main CPU on the R-30iB Plus controller cabinet. When the power disconnect circuit breaker is OFF, power is still present inside the controller. Turn off the power disconnect circuit breaker before you insert a memory card into the memory card interface; otherwise, you could injure personnel.

**⚠ CAUTION**

Do not remove the memory card when the controller is reading or writing to it. Doing so could damage the card and lose all information stored on it.

The memory card can be formatted on the controller, and can be used as a load device to install software.

**NOTE**

Data on all internal file devices such as FR:, RD:, and MD: should be backed up to external file devices such as ATA Flash PC cards.

**NOTE**

The controller formats the card with a sector size of 512 bytes.

The memory card can be formatted and used as an MS-DOS file system. It can be read from and written to the controller and an IBM PC equipped with the proper hardware and software. If the memory card is used as an MS-DOS file system, it should be formatted on the controller.

The controller can read and write memory cards that are formatted with FAT or FAT32 type of formatting (File Access Tables). When a memory card is formatted on the Controller it can be formatted as FAT or FAT32 type.

The FAT32 format (32 Bit FAT) removes a few limitations that are included with FAT. One of these is the limitation that only 255 files can be created in the Root directory. Another is that FAT format type only supports memory cards up to 2 GB in size. This feature is included in the controller to increase the compatibility of the robot controller with other computer systems.

**USB Memory Stick Device on the controller (UD1:), and USB Memory Stick Device on the iPendant (UT1:)**

The controller supports USB Flash memory. It allows you to load or save files. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)*.

**NOTE**

In order to use the UT: device on the iPendant, you must have the USB Port on iPendant option (J957) installed.

**NOTE**

USB Memory sticks cannot be formatted when plugged into the USB connector on the iPendant (UT1:) They must be formatted on a PC or when plugged into the front panel USB connector on the controller.

**NOTE**

USB Memory sticks with any security or encryption features are not supported.

**NOTE**

Memory sticks larger than 2GB can not be formatted FAT, they must be formatted FAT32.

**NOTE**

Generally the larger the size of the device such as a USB memory stick, the slower the access speed and device performance. A USB memory stick should be formatted FAT if possible. Larger memory sticks formatted as FAT32 will work, but will be slower for file operations and startup of the memory stick when it is first inserted.

**NOTE**

You can perform a backup to the UT1: device but you cannot use UT1: to perform a full load or an Auto Update.

**NOTE**

Since USB is a well-established specification, FANUC America Corporation does not qualify specific USB Memory sticks for use with the robot. FANUC America Corporation uses USB Flash drives manufactured by SanDisk® (CRUZER Mini and Micro) to qualify the operation of the USB interface. Therefore, FANUC America Corporation recommends that you use SanDisk® USB Flash Drives, without security or encryption features. Other drives might work properly but are not specifically qualified by FANUC America Corporation.

The USB Memory Stick Device requires a USB interface which is standard on the controller.

The USB memory stick device can be formatted on the controller.

**⚠ CAUTION**

Do not remove the memory stick when the controller is reading or writing to it. Doing so could damage the memory stick and lose all information stored on it.

**Flash File Storage Disk (FR:)**

Flash File Storage Disk is a portion of FROM memory that functions as a separate storage device. Flash file storage disk (FR:) does not require battery backup for information to be retained. You can store the following information on Flash file storage disk:

- Programs
- System variables
- Anything you can save as a file

Due to the nature of FROM, each time you copy or save a file to the FR: there will be a drop in available FR: memory, even if you are working with the same file.

**RAM Disk**

RAM Disk is a portion of Static RAM (SRAM) or DRAM memory that functions as a separate storage device. Any file can be stored on the RAM Disk. RAM Disk files should be copied to an external device for permanent storage.

The location and size of the RAM disk (RD:) depends on the value of the system variable *\$FILE\_MAXSEC*. The default value of *\$FILE\_MAXSEC* depends on the options and tool packages that are installed.

The value in *\$FILE\_MAXSEC* represents the memory size allocated for RD: in 512 byte sectors. For example, a value of -128 means that 64K of memory is allocated in DRAM for RD:.

- If *\$FILE\_MAXSEC > 0*, then RAM disk is defined to be in the PERM pool of SRAM. Because RAM disk is a portion of SRAM, copy all RAM disk files to an external device for permanent storage to prevent losing information due to loss of battery power or system software loading.

SRAM is battery-backed volatile memory. This means that all information in SRAM, including programs, requires battery backup for information to be retained when the controller is turned off and then on again. Teach pendant programs are automatically stored in the TPP pool of SRAM when you write a program.

**⚠ CAUTION**

Data in SRAM can be lost if the battery is removed or loses its charge, or if new system software is loaded on the controller. To prevent loss of data, back up or copy all files to permanent storage devices such as FR: or ATA Flash PC memory cards.

- If *\$FILE\_MAXSEC < 0*, then RAM disk is defined to be in DRAM.

DRAM is non-battery-backed volatile memory. This means that all information in DRAM disappears between power cycles. In effect, DRAM is a temporary device. Information stored in DRAM is lost when you turn off the controller.

**⚠ CAUTION**

Data in DRAM will be lost if you turn off the controller or if the controller loses power. Do not store anything you want to save beyond the next controller power cycle in DRAM, otherwise, you will lose it.

**NOTE**

Volatile means the memory is lost when power is disconnected. Non-volatile memory does not require battery power to retain.

You can store anything that is a file on the RAM Disk. The RAM disk is already formatted for you.

Information stored on RAM disk can be stored as compressed or uncompressed. By default, information is compressed. If you want information to remain uncompressed, you must use the RDU: device designation to indicate that information will be saved to that device in an uncompressed file format.

**FTP Ethernet Device**

An FTP Ethernet device writes and reads files to and from an FTP server such as a PC connected via Ethernet. It is displayed only if FTP client settings have been made on the host communication screen. Refer to the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for more information.

**Memory Device (MD:)**

The memory device (MD:) treats the controller's program memory as if it were a file device. You can access all teach pendant programs, KAREL programs, and KAREL variables loaded in the controller.

The Memory Device is a group of devices (MD:, MDB:, and optionally FMD:) that provide the following:

- MD: provides access to ASCII and binary versions of user setup and programs
- MDB: provides access to binary versions of user setup and programs (similar to **backup - all of the above** on the teach pendant file menu)
- FMD: provides access to ASCII versions of user setup and programs filtered to include only user settable information (for example, internal timers or time system variables changed by the system are not included) making these files useful for detecting user changes.

**Memory Device Binary (MDB:)**

The memory device binary device (MDB:) allows you to copy the same files as provided by the Backup function on the File Menu. This allows you to back up the controller remotely such as from SMON, FTP, or KCL. For example, you could use the MDB: device to copy all teach pendant files (including invisible files) to the memory card (KCL>copy MDB:\*.tp TO mc:).

**Filtered Memory Device (FMD:)**

The Filtered Memory Device option generates text versions of all backup files of user programs and variables that have been changed manually. Included are system and KAREL variables, position and data registers, teach pendant programs, and I/O configuration data.

When logging into the robot FTP server from a remote client you are defaulted into the MD: device. You can navigate to other robot file devices (FR:, RD:, MC:, MDB:, FMD:) using the change directory service in your remote FTP client. At a command line using the cd command where in this example FMD: is the device being used, this might look like:

```
D:\temp>ftp pderob029
Connected to pderob029.frc.com
220 FTP server ready. [xxxxTool Vx.xxP/01]
User <pderob029.frc.com:<none>>:
230 User logged in [NORM].
ftp>cd fmd:
```

```
250 CWD command successful.  
ftp>
```

You can compare these files with previous versions to determine what users or operators have changed. Variables and programs that change without user input are filtered out, and will appear in filter exclusion files.

After the option is installed, it will run automatically whenever you perform an Ethernet backup of the controller from the FMD: device. After you install the Filtered Memory Device option, any of the following filter exclusion files could appear on the FR: device.

 **CAUTION**

Do not delete these files, or filter exclusion data will be lost:

- FR:SVAREEG.DT
- FR:KVAREEG.DT
- FR:POSREEG.DT
- FR:REGEEG.DT
- FR:TPLINEEG.DT

You can view program, variable, or filter exclusion files via KCL. For example:

```
KCL> DIR FMD:*.*
```

**NOTE**

Computer systems that perform periodic backups could be modified to use the FMD: device instead of the MD: device for some compare operations, for example. Contact FANUC for more information.

**FRA:**

There is a special area for Automatic Backup in the controller F-ROM (FRA:). You do not need an external device to use Automatic Backup, but a memory card can also be used.

## 10.2.2 Memory File Devices

The RAM and F-ROM disks allocate files using blocks. Each block is 512 bytes.

The system variable *\$FILE\_MAXSEC* specifies the number of blocks to allocate for the RAM disk. If the specified number is less than zero, the RAM disk is allocated from DRAM. If it is greater than zero, RAM disk is allocated from CMOS RAM. To change the number of blocks to allocate for the RAM disk, perform the following steps from the KCL prompt:

1. Backup all files on the RAM disk. For more information on how to back up files, refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).
2. Enter DISMOUNT RD:

```
KCL>DISMOUNT RD:
```

**3. Enter SET VAR \$FILE\_MAXSEC**

```
KCL>SET VAR  
$FILE_MAXSEC = <new value>
```

**4. Enter FORMAT RD:**

```
KCL>FORMAT RD:
```

All files will be removed from the RAM Disk when the format is performed.

**5. Enter MOUNT RD:**

```
KCL>MOUNT RD:
```

The RAM disk will be reformatted automatically on INIT start.

The F-ROM disk can only be formatted from the BootROM because the system software also resides on F-ROM. The number of blocks available is set by the system. The hardware supports a limited number of read and write cycles, so while the F-ROM disk will function similar to the RAM disk, it does not erase files that have been deleted or overwritten.

After some use, the F-ROM disk will have used up all blocks. At that time, a purge is required to erase the F-ROM blocks which are no longer needed. For more information on purging, refer to the [Section A.16.29, PURGE\\_DEV Built-In Procedure in Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#), or the [Section C.44, PURGE command in Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#)

For more information on memory, refer to [Section 1.3.1, Memory](#).

## 10.2.3 Virtual Devices

KAREL Virtual Devices are similar to DOS subdirectories. For example

- In DOS, to access a file in a subdirectory, you would view FR:\FR1:\>test.kl.
- In KAREL, to access the same file in a virtual device, you would view FR1:test.kl.

The controller supports seven virtual devices. A number, which identifies the virtual device, is appended to the device name (FR1:). [Table 10.2.3](#) shows some of the valid virtual devices available.

**Table 10.2.3 Virtual Devices**

Device Name	Actual Storage
RD:	RAM disk
FR:	F-ROM disk - compressed and uncompressed files
MF:	Refers to files on both RD: and FR:
RD1: - RD7:	RAM disk - compressed and uncompressed files
FR1: - FR7:	F-ROM disk - compressed and uncompressed files
MF1: - MF7:	Refers to files on both the RAM disk and F-ROM disk of the respective virtual device

### Rules for Virtual Devices

The following rules apply to virtual devices.

- A file name on a virtual device is unique. A file could exist on either the RAM or F-ROM disks, but not both. For example: RD: test.kl and FR: test.kl could not both exist.
- A file name could be duplicated across virtual devices. For example: RD: test.kl, RD1: test.kl, and FR2: test.kl could all exist.
- The MF: device name could be used in any file operation to find a file on a virtual device, when the actual storage device is unknown. For example: MF: test.kl finds either RD: test.kl or FR: test.kl.
- When you use the MF: device as a storage device, the RAM disk is used by default when RD: is in CMOS and  $\$FILE\_MAXSEC > 0$ . The F-ROM disk is used by default when RD: is in DRAM and  $\$FILE\_MAXSEC < 0$ . For example: KCL>COPY FILE FLPY:test.kl to MF2 : The file will actually exist on RD2:
- When listing the MF: device directory, all files on the RAM and F-ROM disks are listed. However, only the files in the specified virtual device are displayed.
- If the RD5: directory is specified instead of MF5:, only those files on the RAM disk in virtual device 5 are listed. If the FR3: directory is specified, only those files on the F-ROM disk in virtual device 3 are listed. For example: KCL>DIR RD5 :
- A file could be copied from one virtual device to another virtual device. A file could also be copied from the RAM disk to the F-ROM disk, and vice versa, if the virtual device is different. For example: KCL>COPY RD1: test.kl to FR3 :
- A file could be renamed only within a virtual device and only on the same device. For example: KCL>RENAME FR2: test.kl FR2: example.kl
- A file could be moved within a virtual device from the RAM disk to the F-ROM disk and vice versa, using a special command which is different from copy. For example: KCL>MOVE MF1: test.kl moves test.kl from the F-ROM disk to the RAM disk. KCL>COPY FR1: test.kl TO RD1: test.kl will also move the file from the F-ROM Disk to the RAM Disk. This is because unique file names can only exist on one device. For more information on moving files, refer to the [Section A.13.5, MOVE\\_FILE Built-In Procedure](#) or the [Section C.42, MOVE FILE command](#)
- Formatting the RAM disk, RD: or MF:, clears all the RAM disk files on all the virtual devices. The files on the F-ROM disk remain intact. For example: KCL>FORMAT RD1 : reformats all RAM disk virtual devices (RD: through RD7:). Reformating will cause existing data to be removed.
- Purge erases all blocks that are no longer needed for all the virtual devices. For more information on purging, refer to the [Section A.16.29, PURGE\\_DEV Built-In Procedure](#) or the [Section C.44, PURGE command](#)

## 10.2.4 File Pipes

The PIP: device allows you to access any number of pipe files. This access is to files that are in the controller's memory. This means that the access to these files is very efficient. The size of the files and number of files are limited by available controller memory. This means that the best use of a file pipe is to buffer data or temporarily hold it.

The file resembles a water pipe where data is poured into one end by the writing task and the data flows out the other end into the reading task. This is why the term used is a pipe. This concept is very similar to pipe devices implemented on UNIX, Windows, and Linux.

Files on the pipe device have a limited size but the data is arranged in a circular buffer. This is also called a circular queue. This means that a file pipe of size 8kbytes (this is the default size) will contain the last 8k of data written to the pipe. When the user writes the ninth kilobyte of data to the pipe, the first kilobyte will be overwritten.

Since a pipe is really used to transfer data from one place to another some application will be reading the data out of the pipe. In the default mode, the reader will WAIT until information has been written. Once the data is available in the pipe the read will complete. A KAREL application might use `BYTES_AHEAD` to query the pipe for the amount of data available to read. This is the default read mode.

A second read mode is provided which is called *snapshot*. In this mode the reader will read out the current content of the pipe. Once the current content is read the reader receives an end of the file. This can be applied in an application like a *flight recorder*. This allows you to record information leading up to an event (such as an error) and then retrieve the last set of debug information written to the pipe. Snapshot mode is a read attribute. It is configured using `SET_FILE_ATTR` built-in. By default, the read operation is not in snapshot mode.

Typical pipe applications involve one process writing data to a pipe. The data can be debug information, process parameters or robot positions. The data can then be read out of the pipe by another application. The reading application can be a KAREL program which is analyzing the data coming out of the pipe or it can be KCL or the web server reading the data out and displaying it to the user in ASCII form.

## KAREL Examples

The following apply to KAREL examples.

- Two KAREL tasks can share data through a pipe. One KAREL task can write data to the pipe while a second KAREL task reads from the pipe. In this case the file attribute `ATR_PIPWAIT` can be used for the task that is reading from the pipe. In this case the reading KAREL task will wait on the read function until the write task has finished writing the data. The default operation of the pipe is to return an end of file when there is no data to be read from the pipe.
- A KAREL application might be executing condition handlers at a very fast data rate. In this case it might not be feasible for the condition handler routine to write data out to the teach pendant display screen because this would interfere with the performance of the condition handler. In this case you could write the data to the PIP: device from the condition handler routine. Another KAREL task might read the data from the PIP: device and display it to the teach pendant. In this case the teach pendant display would not be strictly real time. The PIP: device acts as a buffer in this case so that the condition handler can move on to its primary function without waiting for the display to complete. You can also type the file from KCL at the same time the application is writing to it.

PIP: devices are similar to other devices in the following ways:

- The pipe device is similar in some ways to the RD: device. The RD: device also puts the file content in the system memory. The PIP device is different primarily because the pipe file can be opened simultaneously for read and write.
- Similarly to MC: and FR: devices, the PIP: device is used when you want to debug or diagnose real time software. This allows you to output debug information that you can easily view without interfering with the operation that is writing the debug data. This also allows one task to write information that another task can read.
- The function of the PIP: device is similar to all other devices on the controller. This means that all file I/O operations are supported on this device. All I/O functions are supported and work the same except the following: Chdir, Mkdir, and Rmdir.
- The PIP: device is similar to writing directly to a memory card. However, writing to a memory card will delay the writing task while the delay to the PIP: device is much smaller. This means that any code on the controller can use this device. It also has the ability to retain data through a power cycle.

## Rules for PIP: Devices

The following rules apply to PIP: devices:

- The PIP: device can be used by any application or you can specify an associated common option such as KAREL.

- The device is configurable. You can configure how much memory it uses and whether that memory is CMOS (retained) or DRAM (not retained). You are also able to configure the format of the data in order to read out formatted ASCII type data. The device is configured via the PIPE\_CONFIG built-in.

## Installation, Setup and Operation Sequence

In general the PIP: device operates like any other device. A typical operation sequence includes:

```
OPEN myfile ('PIP:/myfile.dat', 'RW')
Write myfile ('Data that I am logging', CR)
Close myfile
```

If you want to be able to access myfile.dat from the Web server, put a link to it on the diagnostic Web page.

The files on the PIP: device are configurable. By default the pipe configuration is specified in the \$PIPE\_CONFIG system variable. The fields listed in [Table 10.2.4](#) have the following meanings:

**Table 10.2.4 System Variable Field Descriptions**

FIELD	DEFAULT	DEFINITION
\$sectors	8	Number of 1024 byte sectors in the pipe.
\$filedata		Pointer to the actual pipe data (not accessible).
\$recordsize	0	Binary record size, zero means its not tracked.
\$auxword	0	Dictionary element if dictionary format or type checksum.
\$memtyp	0	If non zero use CMOS.
\$format	Undefined	Formatting mode: undefined, function, format string or KAREL type.
\$formatter		Function pointer, C format specifier pointer or type code depending on \$format.

Each pipe file can be configured via the pipe\_config built-in. The PIPE\_CONFIG built-in will be called before the pipe file is opened for writing. Refer to [Section A.16.8, PIPE\\_CONFIG Built-In Procedure](#) for more details.

## Operational Examples

The following example writes data from one KAREL routine into a pipe and then reads it back from another routine. These routines can be called from separate tasks so that one task was writing the data and another task can read the data.

**Figure 10.2.4 Program Example**

```
program pipuform
%nolockgroup
var
    pipe, in_file, mcfile, console:file
    record: string[80]
    status: integer
    parm1, parm2: integer
    msg: string[127]
    cmos_flag: boolean
    n_sectors: integer
    record_size: integer
    form_dict: string[4]
```

```

        form_ele: integer
--
--initialize file attributes
routine file_init (att_file :FILE)
begin
    set_file_attr(att_file, ATR_IA)    --force reads to completion
    set_file_attr(att_file, ATR_FIELD)  --force write to completion
    set_file_attr(att_file, ATR_PASSALL) --ignore cr
    set_file_attr(att_file, ATR_UF)     --binary
end file_init
routine write_pipe
begin
    --file is opened
    file_init (pipe)
    open_file pipe ('rw', 'pip:example.dat')
    status = io_status(pipe)
    write console ('Open pipe status:',status,cr)
--  write extra parameters to pipe
    write pipe (msg::8)
    status = io_status(pipe)
end write_pipe
routine read_pipe
var
    record: string[128]
    status: integer
    entry: integer
    num_bytes: integer
begin
    file_init (in_file)
    open_file in_file ('ro', 'pip:example.dat')
    BYTES_AHEAD(in_file, entry, status)
    status = 0
    read in_file (parm1::4)
    status = IO_STATUS(in_file)
    write console ('parm1 read',status,cr)
    write console ('parm1',parm1,cr)
    read in_file (parm2::4)
    status = IO_STATUS(in_file)
    write console ('parm2 read',parm2,status,cr)
end read_pipe
begin
    SET_FILE_ATTR(console, atr_ia, 0) --ATTR_IA is defined in flbt.ke
    OPEN_FILE console ('RW','CONS:')
    if(uninit(msg)) then
        msg = 'Example'
    endif
    if(uninit(n_sectors)) then
        cmos_flag = true
        n_sectors = 16
        record_size = 128
        form_dict = 'test'
        form_ele = 1
    endif
    --
    [in] pipe_name: STRING;name of tag
    [in] cmos_flag: boolean;
    [in] n_sectors: integer;
    [in] record_size: integer;
    [in] form_dict: string;
    [in] form_ele: integer;
    [out] status: INTEGER
    pipe_config('pip:example.dat',cmos_flag, n_sectors,

```

```
        record_size,form_dict,form_ele,status)
write_pipe
read_pipe
close file pipe
close file in_file
end pipuform
```

## 10.3 FILE ACCESS

---

You can access files using the **FILE** and **SELECT** screens on the CRT/KB or teach pendant, or by using KAREL language statements. During normal operations, files will be loaded automatically into the controller. However, other functions could need to be performed.

## 10.4 FORMATTING XML INPUT

---

This feature allows KAREL programs to input data via an XML (eXtended Markup Language) formatted text file. The XML rather than binary format allows the file to be manipulated easily on a PC.

The XML files must follow the most basic XML syntax requirements. These requirements are:

- XML files can have ONLY ONE top level element.
- The start tag must have a matching end tag.
- Empty tags can be represented as <tag parameters/>
- Tags cannot contain special characters such as the set of \*, \$, and [ ]
- They must not contain unprintable characters
- Attributes must be of the form attr="value"
- Special characters are used for the following (outside of tags):
  - < is substituted with &lt;
  - > is substituted with &gt;
  - & is substituted with &amp;
  - “ can be substituted with &quot;
- This feature provides an XML parser and the means for both KAREL and C programmers to easily extract binary data from the text information in an XML file. It does not require the application program to do any parsing of the XML file.

**NOTE**

XML file can have only one top level element. For example,

```
<GRID>
  <TPPROG>
  </TPPROG>
</GRID>
```

is legal. It has one top level element (GRID).

```
<GRID>
</GRID>
<TPPROG>
</TPPROG>
```

is not legal. The master tag can be used to distinguish a GRID file from a password configuration file, for example.

## **10.4.1 Installation Sequence**

This feature consists of KAREL built-ins which provide access to this library for KAREL users. The environment file xml.ev must be on the translator path to translate KAREL programs which reference these built-ins. These built-ins are XML\_ADDTAG, XML\_GETDATA, XML\_REMTAG, XML\_SCAN, and XML\_SETVAR. Refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more information on these built-ins.

## **10.4.2 Example KAREL Program Referencing an XML File**

- Parse the XML file referred to by `xml_name` and return the settings in that file to `xmlstruct`.
- The attribute name-value pairs are returned as strings in `attrnames` and `attrvalues`. It is not required that the data in the XML file be set to a structure. In some applications the name-value pairs are used directly.
- The most efficient XML implementation uses many name-value pairs and only a few tags. It takes the same amount of time to return one name-value pair from a tag as it takes for 32 pairs. Thirty-two tags will take 32 times longer.
- The maximum number of pairs supported is 32.
- There are two different types of XML files. [Figure 10.4.2 \(a\)](#) and [Figure 10.4.2 \(b\)](#) illustrate the two types of tag constructs
- For separated start and end tags ([Figure 10.4.2 \(a\)](#)) the tag processing must be done on the `XML_START` return code.
- For combined start and end tags ([Figure 10.4.2 \(b\)](#)) you cannot provide any text within the tag. KAREL XML processing provides the means to extract this text when required.
- For combined start and end tags ([Figure 10.4.2 \(b\)](#)) the tag processing must be done on the `XML_END` return code.
- The `XML_START` return code needs to set a flag indicating that the tag has been processed.

- The XML\_END return code needs to check to see if processing was already done on the start code and reset the flag.

```
<?xml version="1.0" ?><!-- This is a comment -- >
<xmlstrct_t first="123456" second="7.8910" third="1" fourth="A
string">
Text associated with xmlstrct_t tag
</xmlstrct_t>
```

**Figure 10.4.2 (a) XML File 1 Separated start and end tags**

```
<?xml version="1.0" ?><!-- This is a comment -- >
<xmlstrct_t first="78910" second="12.3456" third="0" fourth="A
string"/>
```

**Figure 10.4.2 (b) XML File 2 Combined start and end tags**

```
<?xml version="1.0" ?>-<!-comment
<GRID rows="16" cols="24" scale="80">
  <xmlstrct_t first="123456" second="78910" third="1" "fourth="A
String">
    special characters < > & "
  </xmlstrct_t>
</GRID>
```

**Figure 10.4.2 (c) XML File 3 GRID tag not registered or processed**

The GRID tag can be in the XML file but not processed by this example program. In general XML tags can be processed by different software. Information is only returned to the KAREL program for tags which are registered by the KAREL program.

```
PROGRAM xmlparse
%COMMENT = 'XML Parse'
%NOPAUSESHT
%NOPAUSE = ERROR + TPENABLE + COMMAND
%NOABORT = ERROR + COMMAND
%NOLOCKGROUP
%NOBUSYLAMP
%ENVIRONMENT xml
%include klerxmlf
CONST
MYXML_CONST = 3
TYPE
  xmlstrct_t = STRUCTURE
    first: INTEGER
    second: REAL
    third: BOOLEAN
    fourth: STRING[20]
  ENDSTRUCTURE
VAR
  xml_name      : string[20]
  tag_name      : string[32]
  text          : array[32] of string[128]
  attrnames    : array[32] of string[32]
  attrvalues   : array[32] of string[64]
  xml_file     : FILE
  status        : INTEGER
  xmlstrct:    xmlstrct_t
  tag_ident:   integer
```

```

func_code: integer
text_idx: integer
numattr: integer
textdone: BOOLEAN
done: BOOLEAN
console: FILE
startdata: BOOLEAN
-----
-
--
-- There are two types of XML file constructs. In one the end tag is
-- embedded in the start tag in the other the end tag is separate. A
-- proper parser must handle both tag constructs
--
-- For the case that the end tag is separate from the start tag
-- (Figure 1.1) the following writes show the sequence of returns:
--Scanned (Rev D) xmlstrct_t 3 100 129015
--Start Tag processing...
--Scanned (Rev D) xmlstrct_t 3 101 129015
--End Tag
--Processed at start tag...
--Scanned (Rev D) xmlstrct_t 0 101 0
--
--For the case where the end tag and start tag are together (Figure
1.2)
--the following writes show the sequences of returns:
--Scanned (Rev D) xmlstrct_t 3 101 129015
--End Tag
--End Tag processing...
--Scanned (Rev D) xmlstrct_t 0 101 0
--
BEGIN
    SET_FILE_ATR(console, ATR_IA, 0) -- ATR_IA is defined in flbt.ke
    OPEN FILE console ('RW', 'CONS:')
    IF UNINIT(xml_name) THEN
        xml_name = 'mc:kl16004.xml'
    ENDIF
    SET_FILE_ATR (xml_file, ATR_XML) -- XML
    CLR_IO_STAT(xml_file)
    OPEN FILE xml_file ('RO', xml_name) -- Open does new operation
    status = IO_STATUS(xml_file)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, 0)
        abort
    ENDIF
    xml_adddtag(xml_file, 'xmlstrct_t', 32, FALSE, MYXML_CONST, status)
    textdone = TRUE
    done = FALSE
    startdata=FALSE
    WHILE (done = FALSE) DO
        xml_scan(xml_file, tag_name, tag_ident, func_code, status)
        if(status = 0) THEN
            done= TRUE
        ENDIF
        WRITE console ('Scanned (Rev D) ', tag_name, ' ', tag_ident,
        ',
                    func_code, ' ', STATUS,' ', CR)
        IF (status = XML_FUNCTION) THEN
            status = 0
            SELECT tag_ident OF

```

```

CASE (MYXML_CONST) :
    SELECT func_code OF
CASE (XML_START) :
    WRITE console ('Start Tag processing...', CR)
        text_idx = 1
        xml_setvar(xml_file, 'k116004', 'xmlstrct', status)
-- Already looked at the attribtues get the text
        xml_getdata(xml_file, numattr, attrnames, attrvalues,
                    text[text_idx], textdone, status)
        startdata = TRUE
CASE (XML_STEND) :
    -- This tag is never returned
    WRITE console ('StEnd Tag', CR)
CASE (XML_END) :
    WRITE console ('End Tag', CR)
    if(startdata = TRUE) THEN
        startdata=FALSE
        WRITE console ('Processed at start tag...', CR)
    ELSE
        WRITE console ('End Tag processing...', CR)
        text_idx = 1
        xml_setvar(xml_file, 'k116004', 'xmlstrct', status)
-- Already looked at the attribtues get the text
        xml_getdata(xml_file, numattr, attrnames, attrvalues,
                    text[text_idx], textdone, status)
    ENDIF
CASE (XML_TXCONT) :
-- Usually the user will do one or the other but not both of
-- these calls
    text_idx = text_idx + 1
    xml_getdata(xml_file, numattr, attrnames, attrvalues,
                text[text_idx], textdone, status)
ELSE:
    ENDSELECT
ELSE:
    ENDSELECT
ELSE
    IF(status <> XML_SCANLIM) THEN
        POST_ERR(status, '', 0, 0)
        done = TRUE
    ENDIF
    ENDIF -- Good status from xml_parse
ENDWHILE
-- This is not required but allows the user to dynamically remove
-- and add tags
xml_remtag(xml_file, 'xmlstrct_t', status)
CLOSE FILE xml_file
status = IO_STATUS(xml_file)
IF status <> 0 THEN
    POST_ERR(status, '', 0, 0)
ENDIF
END xmlparse

```

**Figure 10.4.2 (d) KAREL Program**

Executing this program will extract the attributes first, second, third, and fourth, and their values from the XML file. These values will be set in the variable *xmlstruct* that has fields first, second, third, and fourth. The string variables will also be set to KAREL string variables.

### 10.4.3 Parse Errors

```

XML_TAG_SIZE "Tag too long"
XML_ATTR_SIZE "Attribute too long"
XML_NOSLASH "Invalid use of / character"
XML_INVTAG "Invalid character in tag"
XML_ATTRMATCH "No value for attribute"
XML_TAGMATCH "End tag with no matching start"
XML_INVATTR "Invalid character in attribute"
XML_NOFILE "Cannot find file"
XML_TAGNEST "Tag nesting level too deep"
XML_COMMENT "Error in comment"
XML_BADEXCHAR "Unknown character &xxx;"
XML_TAGNFND "Tag not found"
XML_INVEOF "Unexpected end of file"
XML_SCANLIM "Scan limit exceeded"

```

**NOTE**

XML\_SCANLIM means that the file is too long to be processed in one request. The remedy for this error is to just re-call the XML scan routine as illustrated in the example.

```
XML_FUNCTION "Function code return"
```

### 10.5 MEMORY DEVICE

The Memory device (MD:) treats controller memory programs and variable memory as if it were a file device. Teach pendant programs, KAREL programs, program variables, system variables, and error logs are treated as individual files. This provides expanded functions to communication devices, as well as normal file devices. For example:

1. FTP can load a PC file by copying it to the MD: device.
2. The error log can be retrieved and analyzed remotely by copying from the MD: device.
3. An ASCII listing of teach pendant programs can be obtained by copying \*\*\*.LS from the MD: device.
4. An ASCII listing of system variables can be obtained by copying SYSVARS.VA from the MD: device.

Refer to [Table 10.5 \(a\)](#) for listings and descriptions of files available on the MD device.

**Table 10.5 (a) File Listings for the MD Device**

File Name	Description
ACCNTG.DG	This file shows the system accounting of Operating system tasks.
ACCOFF.DG	This file shows the system accounting is turned off.
AXIS.DG	This file shows the Axis and Servo Status.

<b>File Name</b>	<b>Description</b>
CONFIG.DG	This file shows a summary of system configuration
CONSLOG.DG	This file is an ASCII listing of the system console log.
CONSTAIL.DG	This file is an ASCII listing of the last lines of the system console log.
CURPOS.DG	This file shows the current robot position.
*.DF	This file contains the TP editor default setting.
DIOCFGSV.IO	This file contains I/O configuration information in binary form.
DIOCFGSV.VA	This file is an ASCII listing of DIOCFGSV.IO.
ERRACT.LS	This file is an ASCII listing of active errors.
ERRALL.LS	This file is an ASCII listing of error logs.
ERRAPP.LS	This file is an ASCII listing of application errors.
ERRCOMM .LS	This file shows communication errors.
ERRCURR.LS	This file is an ASCII listing of system configuration.
ERRHIST.LS	This file is an ASCII listing of system configuration.
ERRMOT.LS	This file is an ASCII listing of motion errors.
ERRPWD.LS	This file is an ASCII listing of password errors.
ERRSYS.LS	This file is an ASCII listing of system errors.
ETHERNET	This file shows the Ethernet Configuration.
FRAME.DG	This file shows Frame assignments.
FRAMEVAR.VR	This file contains system frame and tool variable information in binary form.
FRAMEVAR.VA	This file is an ASCII listing of FRAMEVAR.VR.
HIST.LS	This file shows history register dumps.
HISTE.LS	This file is an ASCII listing of general fault exceptions.
HISTP.LS	This file is an ASCII listing of powerfail exceptions.
HISTS.LS	This file is an ASCII listing of servo exceptions.
IOCONFIG.DG	This file shows IO configuration and assignments.
IOSTATE.DG	This file is an ASCII listing of the state of the I/O points.
IOSTATUS.CM	This file is a system command file used to restore I/O.
LOG CONSTAIL.DG	This file is the last line of Console Log.
NUMREG.VA	This file is an ASCII listing of NUMREG.VR.
NUMREG.VR	This file contains system numeric registers.
MACRO.DG	This file shows the Macro Assignment.
MEMORY.DG	This file shows current memory usage.
PORT.DG	This file shows the Serial Port Configuration.
POSREG.VA	This file is an ASCII listing of POSREG.VR.
POSREG.VR	This file contains system position register information.
PRGSTATE.DG	This file is an ASCII listing of the state of the programs.

File Name	Description
RIPELOG.DG	This file contains detailed status information such as the times when robots go ON and OFFLINE, and other diagnostic data. Refer to the <i>Internet Options Setup and Operations Manual (MAROUIN9010171E)</i> or the <i>Ethernet Function OPERATOR'S MANUAL (B-82974EN)</i> for more information.
RIPESTAT.DG	This file contains performance data for you to determine how well the network is performing. Refer to the <i>Internet Options Setup and Operations Manual (MAROUIN9010171E)</i> or the <i>Ethernet Function OPERATOR'S MANUAL (B-82974EN)</i> for more information.
SFTYSIG.DG	This file is an ASCII listing of the state of the safety signals.
STATUS.DG	This file shows a summary of system status
SUMMARY.DG	This file shows diagnostic summaries
SYCLDINT.VA	This file is an ASCII listing of system variables initialized at a Cold start.
SYMOTN.VA	This file is an ASCII listing of motion system variables.
SYNOSAVE.VA	This file is an ASCII listing of non-saved system variables.
SYSFRAME.SV	This file contains \$MNUTOOL, \$MNUFRAME, \$MNUTOOLNUM, and \$MNUFRAMENUM. These variables were in SYSVARS.SV in releases before V7.20.
SYSMACRO.SV	This file is a listing of system macro definitions.
SYSMACRO.VA	This file is an ASCII listing of SYSMACRO.SV.
SYSMAST.SV	This file is a listing of system mastering information.
SYSMAST.VA	This file is an ASCII listing of SYSMAST.SV.
SYSSERVO.SV	This file is a listing of system servo parameters.
SYSSERVO.VA	This file is an ASCII listing of SYSSERVO.SV.
SYSTEM.DG	This file shows a summary of system information
SYSTEM.VA	This file is an ASCII listing of non motion system variables.
SYSVARS.SV	This file is a listing of system variables.
SYSVARS.VA	This file is an ASCII listing of SYSVARS.SV.
SYS****.SV	This file contains application specific system variables.
SYS****.VA	This file is an ASCII listing of SYS****.VA.
TASKLIST.DG	This file shows the system task information.
TESTRUN.DG	This file shows the Testrun Status.
TIMERS.DG	This file shows the System and Program Timer Status.
TPACCN.DG	This file shows TP Accounting Status.
VERSION.DG	This file shows System, Software, and Servo Version Information.
***.PC	This file is a KAREL binary program.
***.VA	This file is an ASCII listing of KAREL variables.
***.VR	This file contains KAREL variables in binary form.
***.LS	This file is an ASCII listing of a teach pendant program.
***.TP	This file is a teach pendant binary program.

File Name	Description
***.TX	This file is a dictionary files.
***.HTM	This file is an HTML web page.
***.STM	This file is an HTML web page using an iPendant Control or Server Side Include.
***.GIF	This file is a GIF image file.
***.JPG	This file is a JPEG image file.

Refer to [Table 10.5 \(b\)](#) for a listing of restrictions when using the MD: device.

**Table 10.5 (b) Testing Restrictions when Using the MD: Device**

File Name or Type	READ	WRITE	DELETE	Comments
***.DG	YES	NO	NO	Diagnostic text file.
***.PC	NO	YES	YES	
***.VR	YES	YES	YES	With restriction of no references.
***.LS	YES	NO	NO	
***.TP	YES	YES	YES	
***.LS	YES	NO	NO	
FFF.DF	YES	YES	NO	
SYS***.SV	YES	YES	NO	Write only at CTRL START.
SYS***.VA	YES	NO	NO	
ERR***.LS	YES	NO	NO	
HISTX.LS	YES	NO	NO	
***REG.VR	YES	YES	NO	
***REG.VA	YES	NO	NO	
DIOCFGSV.IO	YES	YES	NO	Write only at CTRL START.
DIOCFGSV.VA	YES	NO	NO	

# 11 DICTIONARIES AND FORMS

---

Dictionaries and forms are used to create operator interfaces on the teach pendant and CRT/KB screens with KAREL programs.

This chapter includes information about

- Creating user dictionary files, refer to [Section 11.1, CREATING USER DICTIONARIES](#) .
- Creating and using forms, refer to [Section 11.2, CREATING USER FORMS](#) .

In both cases, the text and format of a screen exists outside of the KAREL program. This allows easy modification of screens without altering KAREL programs.

## 11.1 CREATING USER DICTIONARIES

---

A *dictionary file* provides a method for customizing displayed text, including the text attributes (such as blinking, underline, and double wide), and the text location on the screen, without having to re-translate the program.

The following are steps for using dictionaries.

1. Create a formatted ASCII dictionary text file with a .UTX file extension.
2. Compress the dictionary file using the KCL COMPRESS DICT command. This creates a loadable dictionary file with a .TX extension. Once compressed, the .UTX file can be removed from the system. **Only the compressed dictionary (.TX) file is loaded**.
3. Load the compressed dictionary file using the KCL LOAD DICT command or the KAREL ADD\_DICT built-in.
4. Use the KAREL dictionary built-ins to display the dictionary text. Refer to [Section 11.1.12, Accessing Dictionary Elements from a KAREL Program](#) , for more information.

Dictionary files are useful for running the same program on different robots, when the text displayed on each robot is slightly different. For example, if a program runs on only one robot, using KAREL WRITE statements is acceptable. However, using dictionary files simplifies the displaying of text on many robots, by allowing the creation of multiple dictionary files which use the same program to display the text.

### NOTE

Dictionary files are useful in multi-lingual programs.

### 11.1.1 Dictionary Syntax

---

The syntax for a user dictionary consists of one or more dictionary elements. Dictionary elements have the following characteristics:

- A dictionary element can contain multiple lines of information, up to a full screen of information. A user dictionary file has the following syntax:

```
<*comment>
$n<,ele_name><@cursor_pos><&res_word><#chr_code><"Ele_text"><&res_
word>
```

```
<#chr_code><+nest_ele>
<*comment>
<$n+1>
```

- Items in brackets < > are optional.
- \*comment is any item beginning with \*. All text to the end of the line is ignored. Refer to [Section 11.1.9, Dictionary Comment](#).
- \$n specifies the element number. n is a positive integer 0 or greater. Refer to [Section 11.1.2, Dictionary Element Number](#).
- ,ele\_name specifies a comma followed by the element name. Refer to [Section 11.1.3, Dictionary Element Name](#).
- @cursor\_pos specifies a cursor position (two integers separated by a comma.) Cursor positions begin at @1,1. Refer to [Section 11.1.4, Dictionary Cursor Positioning](#).
- &res\_word specifies a dictionary reserved word. Refer to [Section 11.1.6, Dictionary Reserved Word Commands](#).
- "Ele\_text" specifies the element text to be displayed. Refer to [Section 11.1.5, Dictionary Element Text](#).
- +nest\_ele specifies the next dictionary text. Refer to [Section 11.1.8, Nesting Dictionary Elements](#).
- A dictionary element does not have to reside all on one line. Insert a carriage return at any place a space is allowed, except within quoted text. Quoted text must begin and end on the same line.
- Dictionary elements can contain text, position, and display attribute information. [Table 11.1.6](#) lists the attributes of a dictionary element.

## 11.1.2 Dictionary Element Number

A dictionary element number identifies a dictionary element. A dictionary element begins with a \$ followed by the element number. Element numbers have the following characteristics:

- Element numbers begin at 0 and continue in sequential order.
- If element numbers are skipped, the dictionary compressor will add an extra overhead of 5 bytes for each number skipped. Therefore you should not skip a large amount of numbers.
- If you want the dictionary compressor to automatically generate the element numbers sequentially, use a “-” in place of the number. In the following example, the “-” is equated to element number 7.

```
$1
$2
$3
$6
$-
```

## 11.1.3 Dictionary Element Name

Each dictionary element can have an optional element name. The name is separated from the element number by a comma and zero or more spaces. Element names are case sensitive. Only the first 12 characters are used to distinguish element names. For R-30iB Plus, 250 characters are used.

The following are examples of element names:

\$1, KCMN\_SH\_LANG

\$2, KCMN\_SH\_DICT

Dictionary elements can reference other elements by their name instead of by number. Additionally, element names can be generated as constants in a KAREL include file.

## 11.1.4 Dictionary Cursor Positioning

Dictionary elements are displayed in the specified window starting from the current position of the cursor. In most cases, move the cursor to a particular position and begin displaying the dictionary element there.

- The cursor position attribute @ is used to move the cursor on the screen within the window.
- The @ sign is followed by two numbers separated by a comma. The first number is the window row and the second number is the window column.

**For example, on the teach pendant,** the **t\_fu** window begins at row 5 of the **t\_sc** screen and is 10 rows high and 40 columns wide.

- Cursor position @1, 1 is the upper left position of the **t\_fu** window and is located at the **t\_sc** screen row 5 column 1.
- The lower right position of the **t\_fu** window is @10, 40 and is located at the **t\_sc** screen row 15 column 40.

Refer to [Section 7.9.1, USER Menu on the Teach Pendant](#) for more information on the teach pendant screens and windows.

**For example, on the CRT/KB,** the **c\_fu** window begins at row 5 of the **c\_sc** screen and is 17 rows high and 80 columns wide.

- Cursor position @1, 1 is the upper left position of the **c\_fu** window and is located at the **c\_sc** screen row 5 column 1.
- The lower right position of the window is @17, 80 and is located at the **c\_sc** screen row 21, column 80.

Refer to [Section 7.9.2, USER Menu on the CRT/KB](#) for more information on the CRT/KB screens and windows.

The window size defines the display limits of the dictionary elements.

## 11.1.5 Dictionary Element Text

Element text, or quoted text, is the information (text) you want to be displayed on the screen.

- The element text must be enclosed in double quote characters "".
- To insert a back-slash within the text, use \\ (double back-slash.)
- To insert a double-quote within the text, use \" (back-slash, quote.)
- More than one element text string can reside in a dictionary element, separated by reserve words.

Refer to [Section 11.1.6, Dictionary Reserved Word Commands](#) for more information.

- To include the values of KAREL variables in the element text, use the KAREL built-ins. `WRITE_DICT_V` and `READ_DICT_V`, to pass the values of the variables.
- To identify the place where you want the KAREL variables to be inserted, use *format specifiers* in the text.
- A format specifier is the character % followed by some optional fields and then a conversion character. A format specifier has the following syntax:

```
%<-><+><width><.precision>conversion_character<^argument_number>
```

### Format Specifier

- Items enclosed in < > are optional.
- The – sign means left justify the displayed value.
- The + sign means always display the sign if the argument is a number.
- The width field is a number that indicates the minimum number of characters the field should occupy.
- .precision is the . character followed by a number. It has a specific meaning depending upon the conversion character:
- conversion\_characters identify the data type of the argument that is being passed. They are listed in [Table 11.1.6](#).
- ^argument\_number is the ^ (up-caret character) followed by a number.

### Conversion Character

The conversion character is used to identify the data type of the KAREL variable that was passed. [Table 11.1.6](#) lists the conversion characters:

**Table 11.1.5 Conversion Characters**

Character	Argument Type: Printed As
d	INTEGER; decimal number.
o	INTEGER; unsigned octal notation (without a leading zero).
x, X	INTEGER; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
u	INTEGER; unsigned decimal notation.
s	STRING; print characters from the string until end of string or the number of characters given by the precision.
f	REAL; decimal notation of the form <->mmm.ddddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e, E	REAL; decimal notation of the form <->mmm.ddddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g, G	REAL; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal pointer are not printed.
%	No argument is converted; print a %.

- The characters d, o, x, X, and u, can be used with the INTEGER, SHORT, BYTE, and BOOLEAN data types. A BOOLEAN data type is displayed as 0 for FALSE and 1 for TRUE.
- The f, e, E, g, and G characters can be used with the REAL data type.

- The character s is for a STRING data type.

**⚠ CAUTION**

Make sure you use the correct conversion character for the type of argument passed. If the conversion character and argument types do not match, unexpected results could occur.

### Width and Precision

The optional width field is used to fix the minimum number of characters the displayed variable occupies. This is useful for displaying columns of numbers.

Setting a width longer than the largest number aligns the numbers.

- If the displayed number has fewer characters than the width, the number will be padded on the left (or right if the – character is used) by spaces.
- If the width number begins with 0, the field is padded with zeros instead.

The precision has the following meaning for the specified conversion character

- d, o, x, X, and u – The minimum number of digits to be printed. If the displayed integer is less than the precision, leading zeros are padded. This is the same as using a leading zero on the field width.
- s – The maximum number of characters to be printed. If the string is longer than the precision, the remaining characters are ignored.
- f, e, and E – The number of digits to be printed after the decimal point.
- g and G – The number of significant digits.

### Argument Ordering

An element text string can contain more than one format specifier. When a dictionary element is displayed, the first format specifier is applied against the first argument, the second specifier for the second argument, and so on. In some instances, you may need to apply a format specifier out of sequence. This can happen if you develop your program for one language, and then translate the dictionary to another language.

To re-arrange the order of the format specifiers, follow the conversion character with the ^ character and the argument number. As an example,

```
$20, file_message "File %s^2 on device %s^1 not found" &new_line
```

means use the second argument for the first specifier and the first argument for the second specifier.

**⚠ CAUTION**

You cannot re-arrange arguments that are SHORT or BYTE type because these argument are passed differently than other data types. Re-arranging SHORT or BYTE type arguments could cause unexpected results.

## 11.1.6 Dictionary Reserved Word Commands

Reserve words begin with the & character and are used to control the screen. They affect how, and in some cases where, the text is going to be displayed. They provide an easy and self-documenting way of

adding control information to the dictionary. Refer to [Table 11.1.6](#) for a list of the available reserved words.

**Table 11.1.6 Reserved Words**

Reserved Word	Function
&bg_black	Background color black
&bg_blue	Background color blue
&bg_cyan	Background color cyan
&bg_dflt	Background color default
&bg_green	Background color green
&bg_magenta	Background color magenta
&bg_red	Background color red
&bg_white	Background color white
&bg_yellow	Background color yellow
&fg_black	Foreground color black
&fg_blue	Foreground color blue
&fg_cyan	Foreground color cyan
&fg_dflt	Foreground color default
&fg_green	Foreground color green
&fg_magenta	Foreground color magenta
&fg_red	Foreground color red
&fg_white	Foreground color white
&fg_yellow	Foreground color yellow
&clear_win	Clear window (#128)
&clear_2_eol	Clear to end of line (#129)
&clear_2_eow	Clear to end of window (#130)
\$cr	Carriage return (#132)
\$lf	Line feed (#133)
&rev_lf	Reverse line feed (#134)
&new_line	New line (#135)
&bs	Back space (#136)
&home	Home cursor in window (#137)
&blink	Blink video attribute (#138)
&reverse	Reverse video attribute (#139)
&bold	Bold video attribute (#140)
&under_line	Underline video attribute (#141)
&double_wide	Wide video size (#142) (refer to description below for usage)
&standard	All attributes normal (#143)

Reserved Word	Function
&graphics_on	Turn on graphic characters (#146)
&ascii_on	Turn on ASCII characters (#147)
&double_high	High video size (#148) (refer to description below for usage)
&normal_size	Normal video size (#153)
&multi_on	Turn on multi-national characters (#154)

The attributes &normal\_size, &double\_high, and &double\_wide are used to clear data from a line on a screen. However, they are only effective for the line the cursor is currently on. To use these attributes, first position the cursor on the line you want to resize. Then write the attribute, and the text.

- **For the teach pendant**, &double\_high means both double high and double wide are active, and &double\_wide is the same as &normal\_size.
- **For the CRT/KB**, &double\_high means both double high and double wide are active, and &double\_wide means double wide but normal height.

## 11.1.7 Character Codes

---

A character code is the # character followed by a number between 0 and 255. It provides a method of inserting special printable characters, that are not represented on your keyboard, into your dictionary. Refer to [Appendix D, CHARACTER CODES](#), for a listing of the ASCII character codes.

## 11.1.8 Nesting Dictionary Elements

---

The plus + attribute allows a dictionary element to reference another dictionary element from the same dictionary, up to a maximum of five levels. These nested elements can be referenced by element name or element number and can be before or after the current element. When nested elements are displayed, all the elements are displayed in their nesting order as if they are one single element.

## 11.1.9 Dictionary Comment

---

The asterisk character (\*) indicates that all text, to the end of the line, is a comment. All comments are ignored when the dictionary is compressed. A comment can be placed anywhere a space is allowed, except within the element text.

## 11.1.10 Generating a KAREL Constant File

---

The element numbers that are assigned an element name in the dictionary can be generated into a KAREL include file for KAREL programming. The include file will contain the CONST declarator and a constant declaration for each named element.

```
element_name = element_number
```

Your KAREL program can include this file and reference each dictionary element by name instead of number.

To generate a KAREL include file, specify .kl, followed by the file name, on the first line of the dictionary file. The KAREL include file is automatically generated when the dictionary is compressed.

The following would create the file kcmn.kl when the dictionary is compressed.

```
.kl kcmn
$-, move_home, "press HOME to move home"
```

The kcmn.kl file would look as follows

```
-- WARNING: This include file generated by dictionary compressor.
--
-- Include File: kcmn.kl
-- Dictionary file: apkcmneg.utx
--CONST
move_home = 0
```

**NOTE**

If you make a change to your dictionary that causes the element numbers to be re-ordered, you must re-translate your KAREL program to insure that the proper element numbers are used.

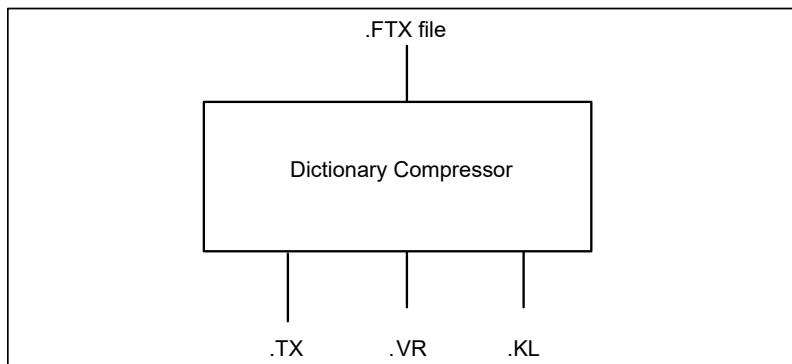
### 11.1.11 Compressing and Loading Dictionaries on the Controller

The KAREL editor can be used to create and modify the user dictionary. When you have finished editing the file, you compress it from the KCL command prompt.

```
KCL> COMPRESS DICT filename
```

Do not include the .UTX file type with the file name. If the compressor detects any errors, it will point to the offending word with a brief explanation of what is wrong. Edit the user dictionary and correct the problem before continuing.

A loadable dictionary with the name filename but with a .TX file type will be created. If you used the .kl symbol, a KAREL include file will also be created. [Figure 11.1.11](#) illustrates the compression process.



**Figure 11.1.11 Dictionary Compressor and User Dictionary File**

Before the KAREL program can use a dictionary, the dictionary must be loaded into the controller and given a dictionary name. The dictionary name is a one to four character word that is assigned to the dictionary when it is loaded. Use the `KCL LOAD DICT` command to load the dictionary.

```
KCL> LOAD DICT filename dictname <lang_name>
```

The optional `lang_name` allows loading multiple dictionaries with the same dictionary name. The actual dictionary that will be used by your program is determined by the current value of `$LANGUAGE`. This system variable is set by the `KCL SET LANGUAGE` command or the `SET_LANG KAREL` built-in. The allowed languages are ENGLISH, JAPANESE, FRENCH, GERMAN, SPANISH, or DEFAULT.

The KAREL program can also load a dictionary. The KAREL built-in `ADD_DICT` is used to load a dictionary into a specified language and assign a dictionary name.

## 11.1.12 Accessing Dictionary Elements from a KAREL Program

Your KAREL program uses either the dictionary name and an element number, or the element name to access a dictionary element. The following KAREL built-ins are used to access dictionary elements:

- `ADD_DICT` - Add a dictionary to the specified language.
- `REMOVE_DICT` - Removes a dictionary from the specified language and closes the file or frees the memory it resides in.
- `WRITE_DICT` - Write a dictionary element to a window.
- `WRITE_DICT_V` - Write a dictionary element that has format specifiers for a KAREL variable, to a window.
- `READ_DICT` - Read a dictionary element into a KAREL STRING variable.
- `READ_DICT_V` - Read a dictionary element that has format specifiers into a STRING variable.
- `CHECK_DICT` - Check if a dictionary element exists.

## 11.2 CREATING USER FORMS

A *form* is a type of dictionary file necessary for creating menu interfaces that have the same look-and-feel as the controller menu interface.

The following are steps for using forms.

1. Create an ASCII form text file with the .FTX file extension.
2. Compress the form file using the KCL COMPRESS FORM command. This creates a loadable dictionary file with a .TX extension and an associated variable file (.VR).
3. Load the form.
  - **From KCL**, use the KCL LOAD FORM command. This will load the dictionary file (.TX) and the associated variable file (.VR).
  - **From KAREL**, use the ADD\_DICT built-in to load the dictionary file (.TX), and the LOAD built-in to load the association variable file (.VR).
4. Use the KAREL DISCTRL\_FORM built-in to display the form text. The DISCTRL\_FORM built-in handles all input operations including cursor position, scrolling, paging, input validation, and choice selections. Refer to the [Section A.4.21, DISCTRL\\_FORM Built-In Procedure](#), [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#).

Forms are useful for programs which require the user to enter data. For example, once the user enters the data, the program must test this data to make sure that it is in an acceptable form. Numbers must be entered with the correct character format and within a specified range, text strings must not exceed a certain length and must be a valid selection. If an improper value is entered, the program must notify the user and prompt for a new entry. Forms provide a way to automatically validate entered data. Forms also allow the program to look as if it is integrated into the rest of the system menus, by giving the operator a familiar interface.

Forms must have the **USER2** menu selected. Forms use the **t\_sc** and **c\_sc** screens for teach pendant and CRT/KB respectively. The windows that are predefined by the system are used for displaying the form text. For both screens, this window is 10 rows high and 40 columns wide. This means that the **&double\_high** and **&double\_wide** attributes are used on the CRT/KB and cannot be changed.

## 11.2.1 Form Syntax

A form defines an operator interface that appears on the teach pendant or CRT/KB screens. A form is a special dictionary element. Many forms can reside in the same dictionary along with other (non-form) dictionary elements.

### NOTE

If your program requires a form dictionary file (.FTX), you do not have to create a user dictionary file (.UTX). You may place your user dictionary elements in the same file as your forms.

To distinguish a form from other elements in the dictionary, the symbol **.form** is placed before the element and the symbol **.endform** is placed after the element. The symbols must reside on their own lines. The form symbols are omitted from the compressed dictionary.

To distinguish a form from other elements in the dictionary, the symbol **.form** is placed before the element and the symbol **.endform** is placed after the element. The symbols must reside on their own lines. The form symbols are omitted from the compressed dictionary.

The following is the syntax for a form:

```
.form <form_attributes>
$n, form_name<@cursor_pos><&res_word>"Menu_title"<&res_work>&new_line
    <@cursor_pos><&res_word>"Menu_label"<&res_word>&new_line
        <@cursor_pos><&res_word><"-Selectable_item"<&res_word>&new_line>
            <@cursor_pos><&res_word><"-%Edit_item"<&res_word>&new_line>
```

```

<@cursor_pos><&res_word><"Non_selectable_text">&res_word>&new_line>
<@cursor_pos><&res_word><"Display_only_item">&res_word>&new_line>
    <^function_key &new_line>
    <?help_menu &new_line>
.endform
<$n,function_key
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"help_label" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    <"Key_name" &new_line>
    "Key_name"
>
<$n,help_menu
    <"Help_text" &new_line>
    <"Help_text" &new_line>
    "Help_text">

```

**Figure 11.2.1 Form Syntax****Restrictions**

- Items in brackets <> are optional.
- Symbols not defined here are standard user dictionary element symbols (\$n, @cursor\_pos, &res\_word, &new\_line).
- form\_attributes are the key words unnumber and unclear.
- form\_name specifies the element name that identifies the form.
- "Menu\_title" and "Menu\_label" specify element text that fills the first two lines of the form and are always displayed.
- "- Selectable\_item" specifies element text that can be cursored to and selected.
- "-%Editable\_item" specifies element text that can be cursored to and edited.
- "Non\_selectable\_text" specifies element text that is displayed in the form and cannot be cursored to.
- "%Display\_only\_item" specifies element text using a format specifier. It cannot be cursored to.
- ^function\_key defines the labels for the function keys using an element name.
- ?help\_menu defines a page of help text that is associated with a form using an element name.
- "Key\_name" specifies element text displayed over the function keys.
- "Help\_label" is the special label for the function key 5. It can be any label or the special word HELP.
- "Help\_text" is element text up to 40 characters long.
- Color attributes can be specified in forms. The iPendant will display the color. The monochrome pendant will ignore the color attributes.

## 11.2.2 Form Attributes

---

Normally, a form is displayed with line numbers in front of any item the cursor can move to. To keep a form from generating and displaying line numbers, the symbol .form unnumber is used.

To keep a form from clearing any windows before being displayed, the symbol .form noclear is used. The symbols noclear and unnumber can be used in any order.

In the following example, MH\_TOOLDEFN is an unnumbered form that does not clear any windows. MH\_APPLIO is a numbered form.

```
.form unnumber noclear
$1, MH_TOOLDEFN
.endform
$2, MH_PORT
$3, MH_PORTFKEY
.form
$6, MH_APPLIO
.endform
```

## 11.2.3 Form Title and Menu Label

---

The menu title is the first element of text that follows the form name. The menu label follows the menu title. Each consists of one row of text in a non-scrolling window.

- **On the teach pendant** the first row of the **full** window is used for the menu title. The second row is used for the menu label.
- **On the CRT/KB** the first row of the **cr05** widow is used for the menu title. The second row is used for the menu label.
- The menu title is positioned at row 3, column 1-21.
- The menu label is positioned at row 4, column 1-40.

Unless the noclear form attribute is specified both the menu title and menu label will be cleared.

The reserved word &home must be specified before the menu title to insure that the cursor is positioned correctly. The reserved word &reverse should also be specified before the menu title and the reserved word &standard should follow directly after the menu title. These are necessary to insure the menu appears to be consistent with the controller menu interface. The reserved word &new\_line must be specified after both the menu title and menu label to indicate the end of the text. The following is an example menu title and menu label definition.

```
.form
$1, mis_form
&home &reverse "Menu Title" &standard &new_line
```

```
"Menu Label" &new_line
.endform
```

If no menu label text is desired, the `&new_line` can be specified twice after the menu title as in the following example.

```
.form
$1,misc_form
&home &reverse " Menu Title" &standard &new_line &new_line
.endform
```

If the cursor position attribute is specified, it is not necessary to specify the `&new_line` reserved word. The following example sets the cursor position for the menu title to row 1, column 2, and the menu label to row 2, column 5.

```
.form
$1,misc_form
@1,2 &reverse "Menu Title" &standard
@2,5 "Menu Label"
.endform
```

## 11.2.4 Form Menu Text

The form menu text follows the menu title and menu label. It consists of an unlimited number of lines that will be displayed in a 10 line scrolling window named **fscr** on the teach pendant and **ct06** on the CRT/KB. This window is positioned at rows 5-14 and columns 1-40. Unless the **noclear** option is specified, all lines will be cleared before displaying the form.

Menu text can consist of the following:

- Selectable menu items
- Edit data items of the following types:
  - INTEGER
  - INTEGER port
  - REAL
  - SHORT (32768 to 32766)
  - BYTE (0 to 255)
  - BOOLEAN
  - BOOLEAN port
  - STRING
  - Program name string
  - Function key enumeration type
  - Subwindow enumeration type
  - Subwindow enumeration type using a variable

- Port simulation
- Non-selectable text
- Display only data items with format specifiers
- Cursor position attributes
- Reserve words or ASCII codes
- Function key element name or number
- Help element name or number

Each kind of menu text is explained in the following sections.

## 11.2.5 Form Selectable Menu Item

---

Selectable menu items have the following characteristics:

- A selectable menu item is entered in the dictionary as a string enclosed in double quotes.
- The first character in the string must be a dash, -. This character will not be printed to the screen. For example,

```
"- Item 1 "
```

- The entire string will be highlighted when the selectable item is the default.
- If a selectable item spans multiple lines, the concatenation character + should be used as the last character in the string. The concatenation character will not be printed to the screen. The attribute &new\_line is used to signal a new line. For example,

```
"- Item 1, line 1 +" &new_line
" Item 1, line 2 "
```

- The automatic numbering uses the first three columns and does not shift the form text. Therefore, the text must allow for the three columns by either adding spaces or specifying cursor positions. For example,

```
"- Item 1 " &new_line
"- Item 2 " &new_line
"- Item 3 "
```

or

```
@3,4"- Item 1 "
@4,4"- Item 2 "
@5,4"- Item 3 "
```

- The first line in the scrolling window is defined as row 3 of the form.
- Pressing enter on a selectable menu item will always cause the form processor to exit with the termination character of ky\_select, regardless of the termination mask setting. The item number selected will be returned.
- Selecting the item by pressing the ITEM hardkey on the teach pendant will only highlight the item. It does not cause an exit.

- Short-cut number selections are not handled automatically, although they can be specified as a termination mask.

## 11.2.6 Edit Data Item

You can edit data items that have the following characteristics:

- Data item is entered in the dictionary as a string enclosed in double quotes.
- The first character in the string must be a dash, -. This character is not printed to the screen.
- The second character in the string must be a %. This character marks the beginning of a format specifier.
- Each format specifier begins with a % and ends with a conversion character. All the characters between these two characters have the same meaning as user dictionary elements.

**NOTE**

You should provide a field width with each format specifier, otherwise a default will be used. This default might cause your form to be mis-aligned.

Table 11.2.6 (a) lists the conversion characters for an editable data item.

**Table 11.2.6 (a) Conversion Characters**

Character	Argument Type: Printed As
d	INTEGER; decimal number.
o	INTEGER; unsigned octal notation (without a leading zero).
x, X	INTEGER; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
u	INTEGER; unsigned decimal notation.
pu	INTEGER port; unsigned decimal notation.
px	INTEGER port; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
f	REAL; decimal notation of the form <->mmm.ddddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e, E	REAL; decimal notation of the form <->m.ddddde+xx or <->m.dddddE+xx, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g, G	REAL; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal pointer are not printed.
h	SHORT; signed short.
b	BYTE; unsigned byte.
B	BOOLEAN; print characters from boolean enumeration string.
P	BOOLEAN port; print characters from boolean port enumeration string.
S	INTEGER or BOOLEAN port; print characters from port simulation enumeration string.

Character	Argument Type: Printed As
k	STRING; print characters from KAREL string until end of string or the number of characters given by the precision.
pk	STRING; print program name from KAREL string until end of string or the number of characters given by the precision.
n	INTEGER; print characters from function key enumeration string. Uses dictionary elements to define the enumeration strings.
w	INTEGER; print characters from subwindow enumeration string. Uses dictionary elements to define the enumeration strings.
v	INTEGER; print characters from subwindow enumeration string. Uses a variable to define the enumeration strings.
%	no argument is converted; print a %.
t	BOOLEAN; print + or - for tree view.

The following is an example of a format specifier:

```
--%5d" or "-%-10s"
```

The form processor retrieves the values from the input value array and displays them sequentially. **All values are dynamically updated.**

#### Edit Data Items: INTEGER, INTEGER Ports, REAL, SHORT, BYTE

- You can specify a range of acceptable values by giving each format specifier a minimum and maximum value allowed "(min, max)". If you do not specify a minimum and maximum value, any integer or floating point value will be accepted. For example,

```
--%3d(1,255)" or "-%10.3f(0.,100000.)"
```

- When an edit data item is selected, the form processor calls the appropriate input routine. The input routine reads the new value (with inverse video active) and uses the minimum and maximum values specified in the dictionary element, to determine whether the new value is within the valid range.
  - If the new value is out of range, an error message will be written to the prompt line and the current value will not be modified.
  - If the new value is in the valid range, it will overwrite the current value.

#### Edit Data Item: BOOLEAN

- The format specifier %B is used for KAREL BOOLEAN values, to display and select menu choice for the **F4** and **F5** function keys. The name of the dictionary element, that contains the function key labels, is enclosed in parentheses and is specified after the %B. For example,

```
--%4B(enum_bool)"
```

The dictionary element defining the function keys should define the FALSE value first (**F5** label) and the TRUE value second (**F4** label). For example,

```
$2,enum_bool
" NO" &new_line
" YES"
```

	YES	NO
--	-----	----

The form processor will label the function keys when the cursor is moved to the enumerated item. The value shown in the field is the same as the function key label except all leading blanks are removed.

### Edit Data Item: BOOLEAN Port

- The format specifier %P is used for KAREL BOOLEAN port values, to display and select menu choices from the **F4** and **F5** function keys. The name of the dictionary element, that contains the function key labels, is enclosed in parentheses and is specified after the %P. For example,

```
"-%3P(enum_bool)"
```

The dictionary element defining the function keys should define the 0 value first (**F5** label) and the 1 value second (**F4** label). For example,

```
$2, enum_bool
" OFF" &new_line
" ON"
```

	ON	OFF
--	----	-----

The form processor will label the function keys when the cursor is moved to the enumerated item. The value shown in the field is the same as the function key label except all leading blanks are removed.

### Edit Data Item: Port Simulation

- The format specifier %S is used for port simulation, to display and select menu choices from the **F4** and **F5** function keys. The name of the dictionary element, that contains the function key labels, is enclosed in parentheses and is specified after the %S. For example,

```
"-%1S(sim_fkey)"
```

The dictionary element defining the function keys should define the 0 value first (**F5** label) and the 1 value second (**F4** label). For example,

```
$-, sim_fkey
" UNSIM" &new_line * F5 key label, port will be unsimulated
"SIMULATE" &new_line * F4 key label, port will be simulated
```

The form processor will label the function keys when the cursor is moved to the enumerated item. The value shown in the field is the same as the function key label except all leading blanks are removed and the value will be truncated to fit in the field width.

### Edit Data Item: STRING

- You can choose to clear the contents of a string before editing it. To do this follow the STRING format specifier with the word `clear`, enclosed in parentheses. If you do not specify (`clear`), the default is to modify the existing string. For example,

```
"-%10k(clear)"
```

### Edit Data Item: Program Name String

- You can use the `%pk` format specifier to display and select program names from the subwindow. The program types to be displayed are enclosed in parenthesis and specified after `%pk`. For example,

```
"-%36pk(1)" * specifies TP programs
"-%36pk(2)" * specifies PC programs
"-%36pk(6)" * specifies TP, PC, VR
"-%36pk(16)" * specifies TP & PC
```

All programs that match the specified type and are currently in memory, are displayed in the subwindow. When a program is selected, the string value is copied to the associated variable.

### Edit Data Item: Function Key Enumeration

- You can use the format specifier `%n` (for enumerated integer values) to display and select choices from the function keys. The name of the dictionary element that contains the list of valid choices is enclosed in parentheses and specified after `%n`. For example,

```
"-%6n(enum_fkey)"
```

The dictionary element defining the function keys should list one function key label per line. If function keys to the left of those specified are not active, they should be set to `""`. A maximum of five function keys can be used. For example,

```
$2,enum_fkey
"" &new_line *Specifies F1 is not active
"JOINT" &new_line *Specifies F2
"LINEAR" &new_line *Specifies F3
"CIRC" *Specifies F4
```

The form processor will label the appropriate function keys when the enumerated item is selected. When a function key is selected, the value set in the integer is as follows:

```
User presses F1, value = 1
User presses F2, value = 2
User presses F3, value = 3
User presses F4, value = 4
```

```
User presses F5, value = 5
```

The value shown in the field is the same as the function key label except all leading blanks are removed.

JOINT	LINEAR	CIRC
-------	--------	------

### Edit Data Item: Subwindow Enumeration

- You can use the format specifier %w (for enumerated integer values) to display and select choices from the subwindow. The name of the dictionary element, containing the list of valid choices, is enclosed in parentheses and specified after %w. For example,

```
"-%8w(enum_sub)"
```

One dictionary element is needed to define each choice in the subwindow. 35 choices can be used. If fewer than 35 choices are used, the last choice should be followed by a dictionary element that contains "\a". The choices will be displayed in 2 columns with 7 choices per page. If only 4 or less choices are used, the choices will be displayed in 1 column with a 36 character width. For example,

```
$2, enum_sub "Option 1"  
$3 "Option 2"  
$4 "Option 3"  
$5 "\a"
```

The form processor will label **F4** as **[CHOICE]** when the cursor is moved to the enumerated item. When the function key **F4**, **[CHOICE]** is selected, it will create the subwindow with the appropriate display. When a choice is selected, the value set in the integer is the number selected. The value shown in the field is the same as the dictionary label except all leading blanks are removed.

### Edit Data Item: Subwindow Enumeration using a Variable

- You can also use the format specifier %v (for enumerated integer values) to display and select choices from the subwindow. However, instead of defining the choices in a dictionary they are defined in a variable. The name of the dictionary element, which contains the program and variable name, is enclosed in parentheses and specified after %v. For example,

```
"-%8v(enum_var)"  
$-, enum_var  
"RUNFORM" &new_line * program name of variable  
"CHOICES" &new_line * variable name containing choices
```

[RUNFORM] CHOICES:ARRAY[6] OF STRING[12] =  
[1] \*uninit\*

[RUNFORM] CHOICES:ARRAY[6] OF STRING[12] =  
[1] \*uninit\*

[RUNFORM] CHOICES:ARRAY[6] OF STRING[12] =  
[1] \*uninit\*

```
[2] 'Red' <= value 1
[3] 'Blue' <= value 2
[4] 'Green' <= value 3
[5] *uninit*
[6] *uninit*
```

### Edit Data Item: Tree View

- The format specifier %t is used to specify a Tree View item. It uses a KAREL BOOLEAN value to determine whether the tree is expanded or collapsed. For example,

```
"-%t"
```

The form processor will change %t to %c. When the BOOLEAN value is FALSE, a + will be displayed using the format specifier and the items following the tree view will not be shown (collapsed state). When the BOOLEAN value is TRUE, a - will be displayed using the format specifier and the items following the tree view will be shown (expanded state). [Table 11.2.6 \(b\)](#) displays some formatting examples.

**Table 11.2.6 (b) Tree View Format**

Format	FALSE value	TRUE value
"-%t"	+	-
"-%2t "	+	-
"-%t Weld schedule:"	+ Weld schedule:	- Weld schedule:

- The KAREL BOOLEAN value will only determine the initial state of the tree view. It is not monitored. When the user selects the tree view item and presses ENTER, the tree view state will toggle and the BOOLEAN value will be set to the resultant value. This state is maintained when the form is exited and reentered. If the KAREL BOOLEAN value is in CMOS or SHADOW, then the state is maintained between power cycles.
- There is a way to refresh the tree view. Passing **ctl\_w** to the form will expand or collapse all tree view items based on their associated KAREL BOOLEAN values. The default item will be maintained. If the default item is no longer shown its tree view item will become the new default item.
- Nesting of tree view items is not allowed. If another %t is found, the current tree view is ended and a new one is started. When a tree view item should be ended without creating a new tree view item, .endtree can be used. For example:

```
"    "-%t Burnback schedule:" &new_line
"        Wire feed """-%6.2f(0.0, 9999.0)" &new_line
"        Trim      """-%6.2f(0.0, 9999.0)" &new_line
.endtree
"    Gas postflow time:" &new_line
```

- If a form is constructed by concatenating multiple forms and an item will be within a tree view, then .tree must be used. The .endtree is optional and only necessary if other items will not be within the tree view.

```
.form
$-,wl_data_cmd &new_line &new_line
.tree
```

```
"      Wire Feed """-%6.2f(0.0, 9999.0)" &new_line
.endform
```

## 11.2.7 Dynamic Forms using Tree View

---

The Tree View can be used to create dynamic forms. It can be used instead of concatenating multiple forms into one. In this case, the Tree View Data Item is used without the leading -. For example,

```
%t" &new_line
"  PC Share Configuration:" &new_line
"  WINS Server:          " "16k" &new_line
"  Client Caching:       " "-%10B(bool_fkey)" &new_line
"  Broadcast Discovery:" "-%10B(bool_fkey)" &new_line
```

When the KAREL BOOLEAN value is FALSE, the items following the tree view will be invisible. When the KAREL BOOLEAN value is TRUE, the items following the tree view will be visible. The tree view line is always invisible. Any other items on its line are also invisible.

The KAREL BOOLEAN value will only determine the initial state of the tree view. It is not monitored. The user will never be able to change the state since the tree view item is invisible.

There is a way to refresh the tree view. Passing **ctl\_w** to the form will expand or collapse all tree view items based on their associated KAREL BOOLEAN values. The default item will be maintained. If the default item is no longer shown, the previous item will become the new default item.

## 11.2.8 Non-Selectable Text

---

Non-selectable text can be specified in the form. These items have the following characteristics:

- Non-selectable text is entered in the dictionary as a string enclosed in double quotes.
- Non-selectable text can be defined anywhere in the form, but must not exceed the maximum number of columns in the window.

## 11.2.9 Display Only Data Items

---

Display only data items can be specified in the form. These items have the following characteristics:

- Display only data items are entered in the dictionary as a string enclosed in double quotes.
- The first character in the string must be a %. This character marks the beginning of a format specifier.
- The format specifiers are the same as defined in the previous section for an edit data item.

## 11.2.10 Cursor Position Attributes

---

Cursor positioning attributes can be used to define the row and column of any text. The row is always specified first. The dictionary compressor will generate an error if the form tries to backtrack to a

previous row or column. The form title and label are on rows 1 and 2. The scrolling window starts on row 3. For example,

```
@3,4 "- Item 1"
@4,4 "- Item 2"
@3,4 "- Item 3" <- backtracking to row 3 not allowed
```

Even though the scrolling window is only 10 lines, a long form can specify row positions that are greater than 12. The form processor keeps track of the current row during scrolling.

## 11.2.11 Form Reserved Words and Character Codes

Reserved words or character codes can be used. Refer to [Table 11.2.11 \(a\)](#) for a list of all available reserved words. However, only the reserved words which do not move the cursor are allowed in a scrolling window. Refer to [for a list of available reserved words for a scrolling window.](#)

**Table 11.2.11 (a) Reserved Words**

Reserved Word	Function
&bg_black	Background color black
&bg_blue	Background color blue
&bg_cyan	Background color cyan
&bg_dflt	Background color default
&bg_green	Background color green
&bg_magenta	Background color magenta
&bg_red	Background color red
&bg_white	Background color white
&bg_yellow	Background color yellow
&fg_black	Foreground color black
&fg_blue	Foreground color blue
&fg_cyan	Foreground color cyan
&fg_dflt	Foreground color default
&fg_green	Foreground color green
&fg_magenta	Foreground color magenta
&fg_red	Foreground color red
&fg_white	Foreground color white
&fg_yellow	Foreground color yellow
&clear_win	Clear window (#128)
&clear_2_eol	Clear to end of line (#129)
&clear_2_eow	Clear to end of window (#130)
\$cr	Carriage return (#132)

Reserved Word	Function
\$lf	Line feed (#133)
&rev_lf	Reverse line feed (#134)
&new_line	New line (#135)
&bs	Back space (#136)
&home	Home cursor in window (#137)
&blink	Blink video attribute (#138)
&reverse	Reverse video attribute (#139)
&bold	Bold video attribute (#140)
&under_line	Underline video attribute (#141)
&double_wide	Wide video size (#142) (refer to description below for usage)
&standard	All attributes normal (#143)
&graphics_on	Turn on graphic characters (#146)
&ascii_on	Turn on ASCII characters (#147)
&double_high	High video size (#148) (refer to description below for usage)
&normal_size	Normal video size (#153)
&multi_on	Turn on multi-national characters (#154)

Table 11.2.11 (b) lists the reserved words that can be used for a scrolling window.

**Table 11.2.11 (b) Reserved Words for Scrolling Window**

Reserved Word	Function
&new_line	New line (#135)
&blink	Blink video attribute (#138)
&reverse	Reverse video attribute (#139)
&bold	Bold video attribute (#140)
&under_line	Underline video attribute (#141)
&standard	All attributes normal (#143)
&graphics_on	Turn on graphic characters (#146)
&ascii_on	Turn on ASCII characters (#147)
&multi_on	Turn on multi-national characters (#154)

## 11.2.12 Form Function Key Element Name or Number

Each form can have one related function key menu. A function key menu has the following characteristics:

- The function key menu is specified in the dictionary with a caret, ^, immediately followed by the name or number of the function key dictionary element. For example,

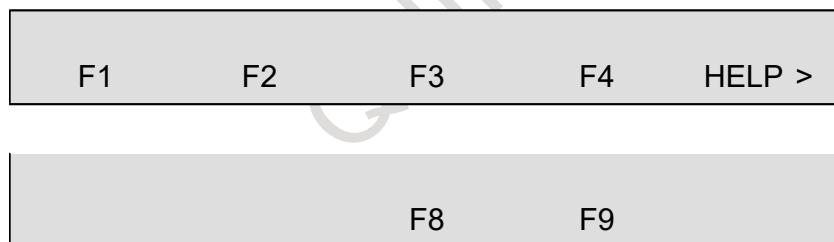
```
^misc_fkey
```

- The dictionary element defining the function keys should list one function key label per line. If function keys to the left of those specified are not active, then they should be set to "". A maximum of 10 function keys can be used. For example,

```
$3,misc_fkey

" F1" &new_line
" F2" &new_line
" F3" &new_line
" F4" &new_line
" HELP >" &new_line
"" &new_line
"" &new_line
" F8" &new_line
" F9" &new_line
```

- The form processor will label the appropriate function keys and return from the routine if a valid key is pressed. The termination character will be set to ky\_f1 through ky\_f10.
- The function keys will be temporarily inactive if an enumerated data item is using the same function keys.
- If function key **F5** is labeled **HELP**, it will automatically call the form's help menu if one exists.



### 11.2.13 Form Function Key Using a Variable

A function key menu can also be defined in a variable. The function key dictionary item will contain the program and variable name, prefixed with an asterisk to distinguish it from function key text. For example,

```
* Specify the function keys in a variable
* whose type is an ARRAY[m] of STRING[n].
$3,misc_fkey
  "*RUNFORM"  &new_line * program name of variable
  "*FKEYS"    &new_line * variable name containing function keys
```

[RUNFORM] FKEYS must be defined as a KAREL string array. Each element of the array should define a function key label.

```
[RUNFORM] FKEYS:ARRAY[10] OF STRING[12] =
[1] ` F1'
[2] ` F2'
[3] ` F3'
[4] ` F4'
[5] ` HELP    >
[6] ` 
[7] ` 
[8] ` F8'
[9] ` F9'
[10]`      >
```

## 11.2.14 Form Help Element Name or Number

---

Each form can have one related help menu. The help menu has the following characteristics:

- A help element name or number is specified in the dictionary with a question mark, ?, immediately followed by the name or number of the help dictionary element. For example,

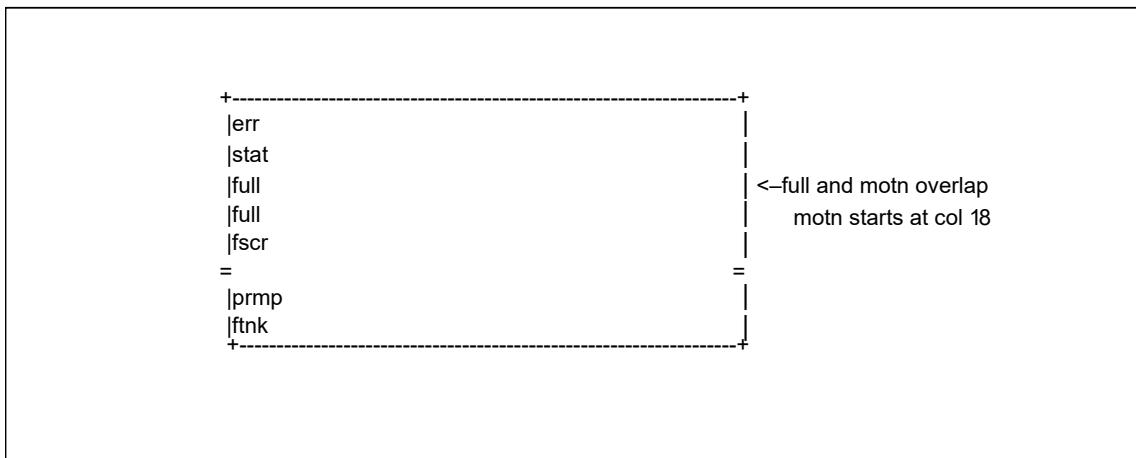
```
?misc_help
```

- The dictionary element defining the help menu is limited to 48 lines of text.
- The form processor will respond to the help key by displaying the help dictionary element in a predefined window. The predefined window is 40 columns wide and occupies rows 3 through 14.
- The help menu responds to the following inputs:
  - Up or down arrows to scroll up or down 1 line.
  - Shifted up or down arrows to scroll up or down 3/4 of a page.
  - Previous, to exit help. The help menu restores the previous screen before returning.

## 11.2.15 Teach Pendant Form Screen

---

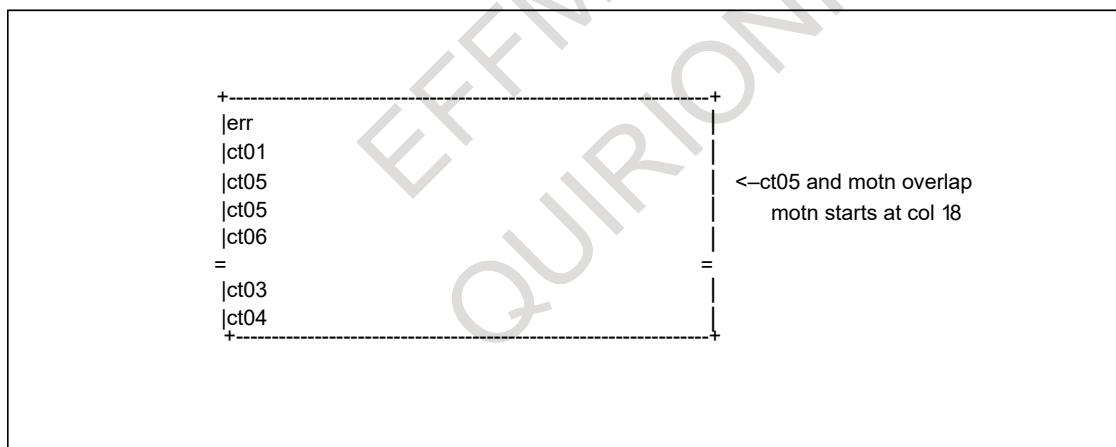
You can write to other active teach pendant windows while the form is displayed. The screen itself is named **tpsc**. shows all the windows attached to this screen. Unless the noclear option is specified, **full**, **fscr**, **prmp**, and **ftnk** windows will be cleared before displaying the form.



**Figure 11.2.15 Teach Pendant Form Screen**

## 11.2.16 CRT/KB Form Screen

You can write to other active CRT/KB windows while the form is displayed. The screen itself is named **ctsc**. All lines in the screen are set to double high and double wide video size. [Figure 11.2.16](#) shows all the windows attached to this screen. Unless the **noclear** option is specified, **ct05**, **ct06**, **ct03**, and **ct04** windows will be cleared before displaying the form.



**Figure 11.2.16 CRT/KB Form Screen**

## 11.2.17 Form File Naming Convention

Uncompressed form dictionary files must use the following file name conventions:

- The first two letters in the dictionary file name can be an application prefix.
- If the file name is greater than four characters, the form processor will skip the first two letters when trying to determine the dictionary name.
- The next four letters must be the dictionary name that you use to load the .TX file, otherwise the form processor will not work.

- The last two letters are optional and should be used to identify the language;
  - EG for ENGLISH
  - JP for JAPANESE
  - FR for FRENCH
  - GR for GERMAN
  - SP for SPANISH
- A dictionary file containing form text must have a .FTX file type, otherwise the dictionary compressor will not work. After it is compressed, the same dictionary file will have a .TX file type instead.

The following is an example of an uncompressed form dictionary file name:

```
MHPALTEG.FTX
```

MH stands for Material Handling, PALT is the dictionary name that is used to load the dictionary on the controller, and EG stands for English.

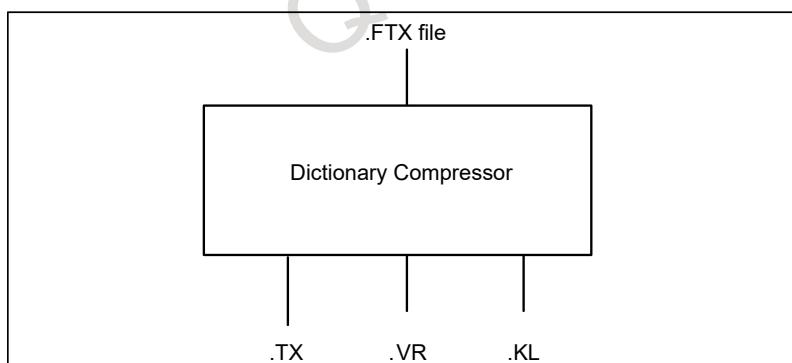
## 11.2.18 Compressing and Loading Forms on the Controller

The form file can only be compressed on the RAM disk RD:. Compressing a form is similar to compressing a user dictionary. From the KCL command prompt, enter:

```
KCL> COMPRESS FORM filename
```

Do not include the .FTX file type. If the compressor detects any errors, it will point to the offending word with a brief explanation of what is wrong. Edit the form and correct the problem before continuing.

Two files will be created by the compressor. One is a loadable dictionary file with the name filename but with a .TX file type. The other will be a variable file with a .VR file type but with the four character dictionary name as the file name. The dictionary name is extracted from filename as described previously. A third file may also be created if you used the .KL symbol to generate a KAREL include file. [Figure 11.2.18](#) illustrates compressing.



**Figure 11.2.18 Dictionary Compressor and Form Dictionary File**

Each form will generate three kinds of variables. These variables are used by the form processor. They must be reloaded each time the form dictionary is recompressed. The variables are as follows:

1. Item array variable - The variable name will be the four-character dictionary name, concatenated with the element number, concatenated with \_IT.
2. Line array variable - The variable name will be the four-character dictionary name, concatenated with the element number, concatenated with \_LN.

3. Miscellaneous variable - The variable name will be the four-character dictionary name, concatenated with the element number, concatenated with \_MS.

The data defining the form is generated into KAREL variables. These variables are saved into the variable file and loaded onto the controller. The name of the program is the dictionary name preceded by an asterisk. For example, Dictionary MHPALTEG.FTX contains:

```
.form unnumber
$1, MH_TOOLDEFN
.endform
$2, MH_PORT
$3, MH_PORTFKEY
.form
$6, MH_APPLIO
.endform
```

As explained in the file naming conventions section, the dictionary name extracted from the file name is PALT. Dictionary elements 1 and 6 are forms. A variable file named PALT.VR is generated with the program name \*PALT. It contains the following variables:

```
PALT1_IT, PALT1_LN, and PALT1_MS
PALT6_IT, PALT6_LN, and PALT6_MS
```

#### **NOTE**

KCL CLEAR ALL will not clear these variables. To show or clear them, you can SET VAR \$CRT\_DEFPROG = '\*PALT' and use SHOW VARS and CLEAR VARS.

The form is loaded using the KCL LOAD FORM command.

```
KCL> LOAD FORM filename
```

The name filename is the name of the loadable dictionary file. After this file is loaded, the dictionary name is extracted from filename and is used to load the variable file. This KCL command is equivalent to

```
KCL> LOAD DICT filename dict_name DRAM
KCL> LOAD VARS dict_name
```

## **11.2.19 Displaying a Form**

The DISCTRL\_FORM built-in is used to display and control a form on the teach pendant or CRT/KB screens. All input keys are handled within DISCTRL\_FORM. This means that execution of your KAREL program will be suspended until an input key causes DISCTRL\_FORM to exit the form. Any condition handlers will remain active while your KAREL program is suspended.

**NOTE**

DISCTRL\_FORM will only display the form if the **USER2** menu is the selected menu. Therefore, use FORCE\_SPMENU(device\_stat, SPI\_TPUSER2, 1) before calling DISCTRL\_FORM to force the **USER2** menu.

The following screen shows the first template in FORM.FTX as displayed on the teach pendant. This example contains four selectable menu items.

RUNFORM	LINE 22	RUNNING
Title here		JOINT 10%
label here		1/5
1 Menu item 1		
2 Menu item 2		
3 Menu item 3		
4 Menu item 4 line 1		
Menu item 4 line 2		
5 Menu item 5		

**Figure 11.2.19 (a) Example of Selectable Menu Items**

The dictionary elements in FORM.FTX, shown in [Figure 11.2.19 \(b\)](#), were used to create the form shown in [Figure 11.2.19 \(a\)](#).

```

*      Dictionary Form File: form.ftx
*
*      Generate form.kl which should be included in your KAREL program
.kl form
.form
$-,form1

&home
&reverse "Title here"
&standard &new_line
    "    label here " &new_line
@3,10    "- Menu item 1 "
@4,10    "- Menu item 2 "
@5,10    "- Menu item 3 "
@6,10    "- Menu item 4 line 1 +"
@7,10    " Menu item 4 line 2 "
@8,10    "- Menu item 5 "
* Add as many items as you wish.
* The form manager will scroll them.
^form1_fkey      * specifies element which contains
                  * function key labels
?form1_help      * element which contains help
.endform
$-,form1_fkey      * function key labels
    "    F1" &new_line
    "    F2" &new_line
    "    F3" &new_line
    "    F4" &new_line
    "    HELP >" &new_line * help must be on F5
    "    F6" &new_line
    "    F7" &new_line
    "    F8" &new_line
    "    F9" &new_line

```

```

        "      F10 >
        * you can have a maximum of 10 function keys labeled
$-, form1_help      * help text
        "Help Line 1" &new_line
        "Help Line 2" &new_line
        "Help Line3" &new_line
        * You can have a maximum of 48 help lines

```

**Figure 11.2.19 (b) Example Form Dictionary for Selectable Menu Items**

The program shown in [Figure 11.2.19 \(c\)](#) was used to display the form shown in [Figure 11.2.19 \(a\)](#).

```

PROGRAM runform
%NOLOCKGROUP
%INCLUDE form           -- allows you to access form element
numbers
%INCLUDE klevccdf
%INCLUDE klevkeys
%INCLUDE klevkmsk
VAR
    device_stat: INTEGER          --tp_panel or crt_panel
    value_array: ARRAY [1] OF STRING [1] --dummy variable for
DISCTRL_FORM
    inact_array: ARRAY [1] OF BOOLEAN     --not used
    change_array: ARRAY [1] OF BOOLEAN     --not used
    def_item: INTEGER
    term_char: INTEGER
    status: INTEGER
BEGIN
    device_stat = tp_panel
    FORCE_SPMENU (device_stat, SPI_TPUSER2, 1)--forces the TP USER2
menu
    def_item = 1 -- start with menu item 1
    --Displays form named FORM1
    DISCTRL_FORM ("FORM", form1, value_array, inact_array,
                  change_array, kc_func_key, def_item, term_char, status)
    WRITE TPERROR (CHR(cc_clear_win))           --clear the TP error
window
    IF term_char = ky_select THEN
        WRITE TPERROR ("Menu item", def_item: :1, 'was selected.')
    ELSE
        WRITE TPERROR ('Func key', term_char: :1, ' was selected.')
    ENDIF
END runform

```

**Figure 11.2.19 (c) Example Program for Selectable Menu Items**

[Figure 11.2.19 \(d\)](#) shows the second template in FORM.FTX as displayed on the CRT/KB (only 10 numbered lines are shown at one time). This example contains all the edit data types.

RUNFORM	LINE 81	RUNNING
Title here		JOINT 10%
<i>label here</i>		
1	Integer:	12345
2	Integer:	1
3	Real:	0.000000
4	Boolean:	TRUE
5	String:	This is a test
6	String:	*****
7	Byte:	10
8	Short:	30
9	DIN[1]:	OFF
10	AIN[1]:	0 S
11	AOUT[2]:	0 U
12	Enum Type:	FINE
13	Enum Type:	Green
14	Enum Type:	Red
15	Prog Type:	MAINTEST
16	Prog Type:	RUNFORM
17	Prog Type:	PRG1
18	Prog Type:	MAINTEST
EXIT		
	F1	F2
		F3
		F4
		F5
	ITEM	PAGE-
		PAGE+
		FCTN
		MENUS
	F6	F7
		F8
		F9
		F10

**Figure 11.2.19 (d) Example of Edit Data Items**

The dictionary elements in FORM.FTX, shown in Figure 11.2.19 (e), were used to create the form shown in Figure 11.2.19 (d).

```
* Dictionary Form File: form.ftx
*
* Generate form.kl which should be included in your KAREL program
.kl form
.form
$-,form2
&home &reverse " Title here" &standard
&new_line
"    label here      "
&new_line
"    Integer:        "    "-%10d"
&new_line
"    Integer:        "    "-%10d(1,32767)"
&new_line
"    Real:          "    "-%12f"
&new_line
"    Boolean:        "    "-%10B(bool_fkey)"
&new_line
"    String:         "    "-%-20k"
&new_line
"    String:         "    "-%12k(clear)"
&new_line
"    Byte:           "    "-%10b"
&new_line
"    Short:          "    "-%10h"
&new_line
"    DIN[1]:         "    "-%10P(dout_fkey)"
```

```

&new_line
"    AIN[1]:          "    "-%10pu"  "  "-%1S(sim_fkey)"
&new_line
"    AOUT[2]:         "    "-%10px"  "  "-%1S(sim_fkey)"
&new_line
"    Enum Type:      "    "-%8n(enum_fkey)"
&new_line
"    Enum Type:      "    "-%6w(enum_subwin)"
&new_line
"    Enum Type:      "    "-%6V(ENUM_VAR)"
&new_line
"    Prog Type:      "    "-%12pk(1)"
&new_line
"    Prog Type:      "    "-%12pk(2)"
&new_line
"    Prog Type:      "    "-%12pk(6)"
&new_line
"    Prog Type:      "    "-%12pk(16)"
&new_line
    ^form2_fkey
.endform
$-, form2_fkey
    EXIT" &new_line
*Allows you to specify the labels for F4 and F5 function keys
$-, bool_fkey
"FALSE"    &new_line  *  F5 key label, value will be set FALSE
"TRUE"     &new_line  *  F4 key label, value will be set TRUE
* Allows you to specify the labels for F4 and F5 function keys
$-, dout_fkey
"OFF"      &new_line  *  F5 key label, value will be set OFF
"ON"       &new_line  *  F4 key label, value will be set ON
*Allows you to specify the labels for F4 and F5 function keys
$-, sim_fkey
" UNSIM"   &new_line  *  F5 key label, port will be unsimulated
"SIMULATE" &new_line  *  F4 key label, port will be simulated
*Allows you to specify the labels for 5 function keys
$-, enum_fkey
"FINE"     &new_line  *  F1 key label, value will be set to 1
"COARSE"   &new_line  *  F2 key label, value will be set to 2
"NOSETTL"  &new_line  *  F3 key label, value will be set to 3
"NODECEL"  &new_line  *  F4 key label, value will be set to 4
"VARDECCEL" &new_line  *  F5 key label, value will be set to 5
*Allows you to specify a maximum of 35 choices in a subwindow
$-, enum_subwin
"Red"      *  value will be set to 1
$-
"Blue"     *  value will be set to 2
$-
"Green"
$-
"Yellow"
$-
"\a"        *  specifies end of subwindow list
* Allows you to specify the choices for the subwindow in a
* variable whose type is an ARRAY[m] of STRING[n].
$-, enum_var

```

```
"RUNFORM"    &new_line      *      program name of variable
"CHOICES"    &new_line      *      Variable name containing choices
```

**Figure 11.2.19 (e) Example Dictionary for Edit Data Items**

The program shown in [Figure 11.2.19 \(f\)](#) was used to display the form in [Figure 11.2.19 \(d\)](#).

```
PROGRAM runform
%NOLOCKGROUP
%INCLUDE form      -- allows you to access form element numbers
%INCLUDE klevccdf
%INCLUDE klevkeys
%INCLUDE klevkmsk
TYPE
    mystruc = STRUCTURE
        byte_var1: BYTE
        byte_var2: BYTE
        short_var: SHORT
    ENDSTRUCTURE
VAR
    device_stat: INTEGER -- tp_panel or crt_panel
    value_array: ARRAY [20] OF STRING [40]
    inact_array: ARRAY [1] OF BOOLEAN
    change_array: ARRAY[1] OF BOOLEAN
    def_item: INTEGER
    term_char: INTEGER
    status: INTEGER
    int_var1: INTEGER
    int_var2: INTEGER
    real_var: REAL
    bool_var: BOOLEAN
    str_var1: STRING[20]
    str_var2: STRING[12]
    struc_var: mystruc
    color_sel1: INTEGER
    color_sel2: INTEGER
    prog_name1: INTEGER[12]
    prog_name2: STRING[12]
    Prog_name3: STRING[12]
    prog_name4: STRING[12]
    choices: ARRAY[5] OF STRING[12]
BEGIN
    value_array [1] = 'int_var1'
    value_array [2] = 'int_var2'
    value_array [3] = 'real_var'
    value_array [4] = 'bool_var'
    value_array [5] = 'str_var1'
    value_array [6] = 'str_var2'
    value_array [7] = 'struc_var.byte_var1'
    value_array [8] = 'struc_var.short_var'
    value_array [9] = 'din[1]'
    value_array [10] = 'ain[1]'
    value_array [11] = 'ain[1]'
    value_array [12] = 'aout[2]'
    value_array [13] = 'aout[2]'
    value_array [14] = '[*system*]$group[1].$termtyp'
    value_array [15] = 'color_sel1'
    value_array [16] = 'color_sel2'
    value_array [17] = 'prog_name1'
    value_array [18] = 'prog_name2'
```

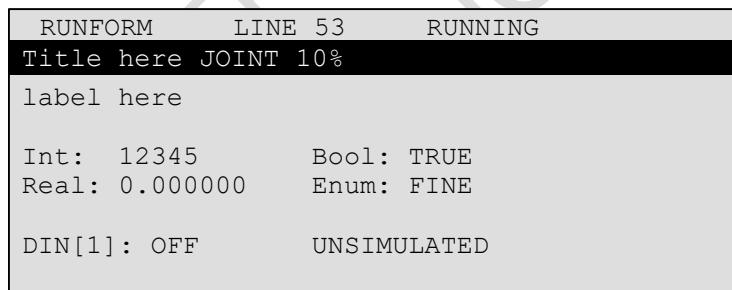
```

value_array [19] = 'prog_name3'
value_array [20] = 'prog_name4'
choices [1] = '' --not used
choices [2] = 'Red' --corresponds to color_sel12 = 1
choices [3] = 'Blue' --corresponds to color_sel12 = 2
choices [4] = 'Green' --corresponds to color_sel12 = 3
choices [5] = 'Yellow' --corresponds to color_sel12 = 4
-- Initialize variables
int_var1 = 12345
-- int_var2 is purposely left uninitialized
real_var = 0
bool_var = TRUE
str_var1 = 'This is a test'
-- str_var = is purposely left uninitialized
struc_var.byte_var1 = 10
struc_var.short_var = 30
color_sel1 = 3 --corresponds to third item of
enum_subwin
color_sel2 = 1
device_stat = crt_panel --specify the CRT/KB for displaying
form
FORCE_SPMENU(device_stat, SPI_TPUSER2,1)
def_item = 1 -- start with menu item 1
DISCTRL_FORM('FORM', form2, value_array, inact_array,
    change_array, kc_func_key, def_item, term_char, status);
END runform

```

**Figure 11.2.19 (f) Example Program for Edit Data Items**

Figure 11.2.19 (g) shows the third template in FORM.FTX as displayed on the teach pendant. This example contains display only items. It shows how to automatically load the form dictionary file and the variable data file, from a KAREL program.

**Figure 11.2.19 (g) Example of Display Only Data Item**

The dictionary elements in FORM.FTX, shown in Figure 11.2.19 (h), were used to create the form shown in Figure 11.2.19 (g).

```

* Dictionary Form File: form.ftx
*
* Generate form.kl which should be included in your KAREL
program
.kl form
.form
$-,form3
    &home &reverse "Title here" &standard      &new_line
    "label here"                                &new_line
    &new_line
    "Int: " "%-10d" " Bool: " "%-10B(bool_fkey)" &new_line
    "Real: " "%-10f" " Enum: " "%-10n(enum_fkey)" &new_line

```

```
"DIN[""%1d"": " "%-10P(dout_fkey)" "%-12S(sim2_fkey)"
*You can have as many columns as you wish without exceeding * 40
columns.
*You can specify blank lines too.
.endform
$-,sim2_fkey
"UNSIMULATED" &new_line * F5 key label, port will be unsimulated
"SIMULATED"   &new_line * F4 key label, port will be simulated
```

**Figure 11.2.19 (h) Example Dictionary for Display Only Data Items**

The program shown in [Figure 11.2.19 \(i\)](#) was used to display the form shown in [Figure 11.2.19 \(g\)](#).

```
PROGRAM runform
%NOLOCKGROUP
%INCLUDE form      -- allows you to access form element numbers
%INCLUDE klevccdf
%INCLUDE klevkeys
%INCLUDE klevkmsk
    device_stat: INTEGER -- tp_panel or crt_panel
    value_array: ARRAY [20] OF STRING [40]
    inact_array: ARRAY [1] OF BOOLEAN -- not used
    change_array: ARRAY[1] OF BOOLEAN -- not used
    def_item: INTEGER
    term_char: INTEGER
    status: INTEGER
    loaded: BOOLEAN
    initialized: BOOLEAN
int_var1: INTEGER
int_var2: INTEGER
real_var: REAL
bool_var: BOOLEAN
BEGIN
-- Make sure 'FORM' dictionary is loaded.
CHECK_DICT('FORM', form3, status)
IF status <> 0 THEN
    WRITE TPPROMPT(CR,'Loading form.....')
    KCL ('CD MF2:',status)           --Use the KCL CD command to
                                     --change directory to MF2:
    KCL ( 'LOAD FORM', status)       --Use the KCL load for
command                                         --to load in the form
    IF status <> 0 THEN
        WRITE TPPROMPT(CR,'loading from failed, STATUS=',status)
        ABORT      --Without the dictionary this program cannot
continue.
    ENDIF
ELSE
    WRITE TPPROMPT (CR,'FORM already loaded.')
ENDIF
    value_array [1] = 'int_var1'
    value_array [2] = 'bool_var'
    value_array [3] = 'real_var'
    value_array [4] = '[*system*]$group[1].$termtyp'
    value_array [5] = 'int_var2'
    value_array [6] = 'din[1]'
    value_array [7] = 'din[1]'
int_var1 = 12345
bool_var = TRUE
real_var = 0
```

```
int_var2 = 1
device_stat = tp_panel
FORCE_SPMENU(device_stat, SPI_TPUSER2,1)
def_item = 1 -- start with menu item 1
DISCTRL_FORM('FORM', form3, value_array, inact_array,
              change_array, kc_func_key, def_item, term_char, status);
END runform
```

**Figure 11.2.19 (i) Example Program for Display Only Data Items**

EFFMAN  
QUIRIONR

# 12 SOCKET MESSAGING

---

The User Socket Messaging Option gives you the benefit of using TCP/IP socket messaging from KAREL.

Socket Messaging enables data exchange between networked robots and a remote PC with LINUX, or a UNIX workstation. A typical application of Socket Messaging might be a robot running a KAREL program that sends process information to a monitoring program on the remote PC. The combination of PC-Interface option on the robot and PC-Developers Kit on the PC is recommended for data exchange between the robot and a Windows-based PC.

Socket Messaging uses the TCP/IP protocol to transfer raw data, or data that is in its original, unformatted form across the network. Commands and methods that Socket Messaging uses to transfer data are part of the TCP/IP protocol. Since Socket Messaging supports client and server tags, applications requiring timeouts, heartbeats, or data formatting commands can provide these additional semantics at both the client and server (application) sides of the socket messaging connection.

## 12.1 SYSTEM REQUIREMENTS

---

This section contains information about the compatibility of socket messaging with some typical network software, transmission protocols, and interface hardware.

### 12.1.1 Software Requirements

---

Socket Messaging is compatible with all other Internet Options including DNS, FTP, Web Server, and Telnet.

#### NOTE

Client and Server tags are shared between Socket Messaging and other supported protocols, such as FTP. For example, a tag can be set for either FTP operation, or for SM (Socket Messaging) operation.

### 12.1.2 Hardware Requirements

---

Socket Messaging is compatible with all network hardware configurations that use the TCP/IP network protocol. Some of these network hardware configurations include Ethernet, serial PPP connections and PPP modem connections.

## 12.2 CONFIGURING THE SOCKET MESSAGING OPTION

---

In order to use Socket Messaging, you need to configure the following network hardware and software parameters:

- On the server,
  - The port you want to use for socket messaging
- On the client,
  - The IP address or name of your server
  - The port on the server that you want to use for socket messaging.

Use [Section 12.2.1, Setting up a Server Tag](#) to set up a Socket Messaging Server Tag. Use [Section 12.2.2, Setting up a Client Tag](#) to set up a Socket Messaging Client Tag.

**NOTE**

The server port at which the server listens on should match the port the client tries to connect on.

## 12.2.1 Setting up a Server Tag

You need configure the server tags you want to use for socket messaging. Use [Section 12.2.1.1, Setting up a Server Tag](#) to set up your server tags.

**NOTE**

If the server tags you want to use are being used by a network protocol other than TCP/IP, you need to undefine the tags before they can be used for socket messaging. After making sure the tag you want to use is not critical to another component of your network, you must undefine the tag.

### 12.2.1.1 Setting up a Server Tag

**Before you begin**

- The tag you want to set up is not configured to be used by another device on your network.

**Procedure**

1. Cold start the controller.
  - a) **On the teach pendant**, press and hold the **SHIFT** and **RESET** keys. Or, **on the operator panel**, press and hold **RESET**.
  - b) While still pressing **SHIFT** and **RESET** on the teach pendant (or **RESET** on the operator panel), turn on the power disconnect circuit breaker.
  - c) Release all of the keys.
2. On the teach pendant, press **MENU**.
3. Select **SETUP**.
4. Press **F1, [TYPE]**.
5. Select **Host Comm**.

6. Press **F4, [SHOW]**.
7. Choose **Servers**.
8. Move the cursor to the tag you want set up for Socket Messaging, and press **F3, DETAIL**. You will see screen similar to the following.

SETUP Tags

Tag S3:

Comment:	*****
Protocol Name:	*****
Current State:	UNDEFINED
Startup State:	
Server IP/Hostname:	*****
Remote Path/Share:	*****
Port:	*****
Inactivity Timeout:	15 min
Username:	anonymous
Password	*****

9. Move the cursor to **Protocol name**, and press **F4, [CHOICE]**.
10. Select **SM**.
11. Move the cursor to **Startup State**, and press **F4, [CHOICE]**.
12. Select **START**.
13. Press **F2, [ACTION]**.
14. Select **DEFINE**.
15. Press **F2, [ACTION]**.
16. Select **START**.
17. Set the system variable:
  - a) Press **MENU**.
  - b) Select **NEXT**.
  - c) Select **SYSTEM**, and press **F1, [TYPE]**.
  - d) Select **Variables**.
  - e) Move the cursor to **\$HOSTS\_CFG**, and press **ENTER**.
  - f) Move the cursor to the structure corresponding to the tag selected in [Step 8](#) . For example, if you are setting up tag S3, move the cursor to structure element [3], as shown in the following screen.

SYSTEM Variables

\$HOSTS_CFG	
1 [1]	HOST_CFG_T
2 [2]	HOST_CFG_T

3	[3]	HOST_CFG_T
4	[4]	HOST_CFG_T
5	[5]	HOST_CFG_T
6	[6]	HOST_CFG_T
7	[7]	HOST_CFG_T
8	[8]	HOST_CFG_T

- g) Press **ENTER**. You will see a screen similar to the following.

```
SYSTEM Variables
$HOSTS_CFG[3]
 1 $COMMENT          *uninit*
 2 $PROTOCOL         'SM'
 3 $PORT              *uninit*
 4 $OPER                3
 5 $STATE              3
 6 $MODE              *uninit*
 7 $REMOTE             *uninit*
 8 $REPERRS            FALSE
 9 $TIMEOUT            15
10 $PATH              *uninit*
11 $STRT_PATH          *uninit*
12 $STRT_REMOTE        *uninit*
13 $USERNAME           *uninit*
14 $PWRD_TIMOUT       0
15 $SERVER_PORT         0
```

- h) Move the cursor to **\$SERVER\_PORT**. Type in the name of the TCP/IP port you want to use for socket messaging. The server tag is now ready to use from a KAREL program.

## 12.2.2 Setting up a Client Tag

You need to configure the client tags you want to use for socket messaging. Use [Section 12.2.2.1, Setting up a ClientTag](#) to set up your client tags. You can also use [Section 12.2.2.1, Setting up a ClientTag](#) to undefine tags.

### NOTE

If the client tags you want to use are being used by a network protocol other than TCP/IP, you need to undefine the tags before they can be used for socket messaging.

### 12.2.2.1 Setting up a ClientTag

#### Before you begin

- The tag you want to set up is not configured to be used by another device on your network.

#### Procedure

- Cold start the controller.

- a) **On the teach pendant**, press and hold the **SHIFT** and **RESET** keys. Or, **on the operator panel**, press and hold **RESET**.
  - b) While still pressing **SHIFT** and **RESET** on the teach pendant (or **RESET** on the operator panel), turn on the power disconnect circuit breaker.
  - c) Release all of the keys.
2. On the teach pendant, press **MENU**.
  3. Select **SETUP**.
  4. Press **F1, [TYPE]**.
  5. Select **Host Comm**.
  6. Press **F4, [SHOW]**.
  7. Choose **Clients**.
  8. Move the cursor to the tag you want set up for Socket Messaging, and press **F3, DETAIL**. You will see screen similar to the following.

#### SETUP Tags

Tag C3:

Comment:	*****
Protocol Name:	*****
Current State:	UNDEFINED
Startup State:	
Server IP/Hostname:	*****
Remote Path/Share:	*****
Port:	*****
Inactivity Timeout:	15 min
Username:	anonymous
Password	*****

9. Move the cursor to the **Protocol Name** item, and press **F4, [CHOICE]**.
10. Select **SM**.
11. Move the cursor to the **Startup State** item, press **F4, [CHOICE]**, and choose **DEFINE**.
12. Move the cursor to the **Server IP/Hostname** item, and press **ENTER**.
13. Type in hostname or IP address the of the remote host server you want to use for socket messaging.

#### NOTE

If you are not using DNS, you must add the remote host and its IP address into the host entry table.

14. Press **F2, [ACTION]**, and select **DEFINE**.

**15.** Set the system variable:

- a) Press **MENU**.
- b) Select **NEXT**.
- c) Select **SYSTEM**, and press **F1, [TYPE]**.
- d) Select **Variables**.
- e) Move the cursor to **\$HOSTC\_CFG**, and press **ENTER**.
- f) Move the cursor to the structure corresponding to the tag selected in [Step 8](#). For example, if you are setting up tag C3, move the cursor to structure element [3], as shown in the following screen.

```
SYSTEM Variables
$HOSTC_CFG
 1 [1]           HOST_CFG_T
 2 [2]           HOST_CFG_T
 3 [3]           HOST_CFG_T
 4 [4]           HOST_CFG_T
 5 [5]           HOST_CFG_T
 6 [6]           HOST_CFG_T
 7 [7]           HOST_CFG_T
 8 [8]           HOST_CFG_T
```

- g) Press **ENTER**. You will see a screen similar to the following.

```
SYSTEM Variables
$HOSTC_CFG[3]
 1 $COMMENT      *uninit*
 2 $PROTOCOL     'SM'
 3 $PORT          *uninit*
 4 $OPER          3
 5 $STATE         3
 6 $MODE          *uninit*
 7 $REMOTE        *uninit*
 8 $REPERRS       FALSE
 9 $TIMEOUT       15
10 $PATH          *uninit*
11 $STRT_PATH    *uninit*
12 $STRT_REMOTE  *uninit*
13 $USERNAME      *uninit*
14 $PWRD_TIMOUT  0
15 $SERVER_PORT   0
```

- h) Move the cursor to **\$SERVER\_PORT**. Type in the name of the TCP/IP server port you want to use for socket messaging. The client tag is now ready to use from a KAREL program.

## 12.3 SOCKET MESSAGING AND KAREL

Socket messaging is an integrated component of KAREL. When you use socket messaging functions and utilities from a KAREL program, the syntax is similar to other file read and write operations, except that you need to establish a network connection when you use socket messaging functions and utilities.

The following KAREL socket messaging functions and utilities enable the server to establish a connection with a remote host on your network. There are several KAREL program samples in this section that provide examples of how these functions and utilities can be used with KAREL file read and

write functions and utilities to write a complete Socket Messaging KAREL client or a server program or application. The Environment flbt statement is required to use any of the listed built-ins (%ENVIRONMENT flbt).

### **12.3.1 MSG\_CONNECT(string, integer)**

---

MSG\_CONNECT needs to be called before any tag can be used for socket messaging.

The first parameter of this command contains the tag name (\$1 :, for example) and the second parameter is an integer that will contain the status of the operation. If you are using this command to connect to a server tag, this command will return a status value only after a remote client device has established a connection with this server tag.

If you are using this command to connect to a client tag, this command will return a status value only if the remote server is attempting to accept the connection. If the connection was successful, the command will return a value indicating a successful connection was made. If the connection was not successful, the command will return a value indicating that a connection error has occurred.

During a socket messaging session, you must use MSG\_DISCO to close the socket connection with a client or server tag before any subsequent attempts to connect to the same client or server tag can be made using MSG\_CONNECT.

### **12.3.2 MSG\_DISCO(string, integer)**

---

MSG\_DISCO is used to close socket messaging connections. If a connection is lost, perhaps because a READ or WRITE error occurred when the remote server terminated a socket messaging connection, you will need to use MSG\_DISCO to close the connection to the remote server. In this case, MSG\_DISCO must be used to close the connection at the client side before MSG\_CONNECT can be used to establish another connection to the remote server.

The first parameter of this command contains the tag name (\$1 :, for example) and the second parameter is an integer value that indicates the status of closing the connection on the client side.

### **12.3.3 MSG\_PING(string, integer)**

---

MSG\_PING is a utility command used to check network connections with a remote host, so that you can determine if it is currently connected to the network. The MSG\_PING command sends *ping packets* to the remote host and waits for a reply. (Ping packets are chunks of data that are transferred between hosts on a network.) If there is no reply from the remote host, this usually means that you will not be able to use other network protocols like FTP, TELNET, or Socket Messaging to connect to that host. If you have attempted without success to use Socket Messaging to connect to a remote host, the MSG-PING utility is a good place to start in trying to diagnose the problem.

The first parameter of this command contains the name of the remote host to ping. If you are not using DNS on your network, the host name and IP address of the remote host will have to exist in the Host Entry table.

### 12.3.4 Exchanging Data during a Socket Messaging Connection

After you have successfully established a socket messaging connection, you can use KAREL commands to exchange data between connected devices. KAREL has several commands that can be used for data exchange operations:

- OPEN FILE
- WRITE
- READ
- BYTES\_AHEAD

## 12.4 NETWORK PERFORMANCE

Performance of socket messaging on your network will vary depending upon the number of devices connected to the network, the number of applications being run on the controller, the network cabling configuration, and number of *hops* that the message will have to make to reach its destination device.

#### NOTE

Hops is a term that indicates the number of routers between the source host and destination host. In general, the fewer the number of hops the data makes from router to router, the faster the data is transmitted between the source and destination hosts. Data transfer is fastest between hosts on the same network.

### 12.4.1 Guidelines for a Good Implementation

Use the following guidelines when implementing a solution for any application using socket messaging.

- You must not transfer small data separately, instead gather the data and transfer as a larger packet. This applies to both read and write.
- You must understand that the rate of data does not flood the remote side.
- You must understand that there are other nodes on the Ethernet network so performance cannot be guaranteed.
- You must understand that other applications on robot also use TCP/IP and performance can be affected.

## 12.5 PROGRAMMING EXAMPLES

This section contains programming examples for a KAREL socket messaging client, and a KAREL socket messaging server. There is also a UNIX-based ANSI C example for a loopback client application, which assumes that you have access to a UNIX-compatible ANSI C compiler, and a basic knowledge of programming in the ANSI C language.

**NOTE**

The KAREL examples assume the appropriate tags (C2 for client and S3 for Server) have been set up for socket messaging using [Section 12.2.1.1, Setting up a Server Tag](#) and [Section 12.2.2.1, Setting up a ClientTag](#).

## 12.5.1 A KAREL Client Application

[Figure 12.5.1](#) provides code for a basic KAREL client application that can be used to establish a socket messaging connection to a remote host, which could be the KAREL server socket messaging application shown in [Figure 12.5.2](#).

```
-- This material is the joint property of FANUC America
-- Corporation and FANUC LTD Japan, and must be returned to
-- either FANUC America Corporation or FANUC LTD Japan
-- immediately upon request. This material and the information
-- illustrated or contained herein may not be reproduced,
-- copied, used, or transmitted in whole or in part in any way
-- without the prior written consent of both FANUC America
-- Corporation and FANUC

-- All Rights Reserved
-- Copyright (C) 2000
-- FANUC America Corporation
-- FANUC LTD Japan

-- Karel is a registered trademark of
-- FANUC America Corporation
+
-- Program: loopcl.kl - Program for TCP Messaging
--
-- Description:
--
-- This program serves as an example on how to use TCP messaging
-- and write a client Karel program.
--
-- Authors: FANUC America Corporation
--          3900 West Hamlin
--          Rochester Hills, MI 48309
--
-- Modification history:
--

-----  
PROGRAM loopcl
%STACKSIZE = 4000
%NOLOCKGROUP
%NOPAUSE=ERROR+COMMAND+TPENABLE
%ENVIRONMENT uif
%ENVIRONMENT sysdef
%ENVIRONMENT memo
%ENVIRONMENT kclop
%ENVIRONMENT bynam
%ENVIRONMENT fdev
%ENVIRONMENT flbt
%INCLUDE klevccdf
%INCLUDE klevkeys
```

```
%INCLUDE klevkmsk
-----
VAR
    file_var : FILE
    tmp_int : INTEGER
    tmp_str : STRING[128]
    status : INTEGER
    entry : INTEGER
    loop1 : BOOLEAN
-----
BEGIN
    SET_FILE_ATR(file_var, ATR_IA)
    SET_VAR(entry,
'*SYSTEM*', '$HOSTC_CFG[2].$SERVER_PORT', 59002, status)
    -- Connect the tag
    WRITE('Connecting..',cr)
    MSG_CONNECT('C2:',status)
    WRITE(' Connect Status = ',status,cr)
    loop1 = TRUE
    IF status = 0 THEN
        WHILE loop1 = TRUE DO
            WRITE('Opening File..',cr)
            OPEN FILE file_var('rw','C2:')
            status = IO_STATUS(file_var)
            IF status = 0 THEN
                FOR tmp_int = 1 TO 100 DO
                    tmp_str = '0123456789012345'
                    WRITE file_var(tmp_str::10)
                    WRITE('Wrote 126 Bytes',cr)
                    IF status <> 0 THEN
                        WRITE('Loop Test Fails',cr)
                        loop1 = FALSE
                        tmp_int = 100
                    ELSE
                        WRITE('Read 126 Bytes',cr)
                        READ file_var(tmp_str::10)
                        status = IO_STATUS(file_var)
                        WRITE('Read Status ',status,cr)
                    ENDIF
                ENDFOR
                WRITE('Closed File',cr)
                CLOSE FILE file_var
            ELSE
                WRITE('Error Opening File',cr)
                loop1 = FALSE
           ENDIF
        ENDWHILE
        WRITE('Disconnecting..',cr)
        MSG_DISCO('C2:',status)
        WRITE('Done.',cr)
    ENDIF
END loop1
```

**Figure 12.5.1 A KAREL Client Application**

## 12.5.2 A KAREL Server Application

[Figure 12.5.2](#) provides code for a basic KAREL server application that can be used to host a socket messaging connection made by a remote client, which could be the KAREL client socket messaging application shown in [Figure 12.5.1](#).

```
-- This material is the joint property of FANUC America
-- Corporation and FANUC LTD Japan, and must be returned to
-- either FANUC America Corporation or FANUC LTD Japan
-- immediately upon request. This material and the information
-- illustrated or contained herein may not be reproduced,
-- copied, used, or transmitted in whole or in part in any way
-- without the prior written consent of both FANUC America
-- Corporation and FANUC.

-- All Rights Reserved
-- Copyright (C) 2000
-- FANUC America Corporation
-- FANUC LTD Japan
-- Karel is a registered trademark of
-- FANUC America Corporation
-- +
-- Program: tcperv3.kl - Program for TCP Messaging

-- Description:

-- This program serves as an example on how to use TCP messaging
-- and write a server Karel program.

-- Authors: FANUC America Corporation
-- 3900 West Hamlin
-- Rochester Hills, MI 48309

-- Modification history:

-----
PROGRAM tcperv3
%STACKSIZE = 4000
%NOLOCKGROUP
%NOPAUSE=ERROR+COMMAND+TPENABLE
%ENVIRONMENT uif
%ENVIRONMENT sysdef
%ENVIRONMENT memo
%ENVIRONMENT kclop
%ENVIRONMENT bynam
%ENVIRONMENT fdev
%ENVIRONMENT flbt
%INCLUDE klevccdf
%INCLUDE klevkeys
%INCLUDE klevkmsk
-----

VAR
  file_var : FILE
  tmp_int : INTEGER
  tmp_int1 : INTEGER
  tmp_str : STRING[128]
  tmp_str1 : STRING[128]
  status : INTEGER
  entry : INTEGER
```

```

-----
BEGIN
    SET _FILE_ATR(file_var, ATR_IA)
    -- set the server port before doing a connect
    SET_VAR(entry,
'*SYSTEM*', '$HOSTS_CFG[3].$SERVER_PORT', 59002, status)
    WRITE('Connecting..',cr)
    MSG_CONNECT('S3:',status)
    WRITE(' Connect Status = ',status,cr)
    IF status = 0 THEN
        -- Open S3:
        WRITE ('Opening',cr)
        FOR tmp_int1 = 1 TO 20 DO
            OPEN FILE file_var ('rw','S3:')
            status = IO_STATUS(file_var)
            WRITE (status,cr)
            IF status = 0 THEN
                -- write an integer
                FOR tmp_int = 1 TO 1000 DO
                    WRITE('Reading',cr)
                    -- Read 10 bytes
                    BYTES_AHEAD(file_var, entry, status)
                    WRITE(entry, status, cr)
                    READ file_var (tmp_str:::10)
                    status = IO_STATUS(file_var)
                    WRITE (status, cr)
                    -- Write 10 bytes
                    WRITE (tmp_str:::10, cr)
                    status = IO_STATUS(file_var)
                    WRITE (status, cr)
                ENDFOR
                CLOSE FILE file_var
            ENDIF
        ENDFOR
        WRITE('Disconnecting..',cr)
        MSG_DISCO('S3:',status)
        WRITE('Done.',cr)
    ENDIF
END tcpserv3

```

**Figure 12.5.2 KAREL Server Application**

### **12.5.3 ANSI C Loopback Client Example**

Figure 12.5.3 provides an example of a UNIX-based loopback client that can be used to establish a connection with a remote host.

```

/* BSD Standard Socket Programming Example - UNIX */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define SERV_TCP_PORT 59002
#define SERV_HOST_ADDR "199.5.148.56"

```

```

#define MAXLINE 512
int written(int fd, char *ptr, int nbytes);
int readline(int fd, char *ptr, int maxlen);
void str_cli(int sockfd);
char *pname;
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in serv_addr;

    pname = argv[0];

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("Client: Can't Open Stream Socket\n");
    }

    printf("Client: Connecting...\n");

    if(connect(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr))<0){
        printf("Client: Can't Connect to the server\n");
    }
    else{
        str_cli(sockfd);
    }
    exit(0);
}

void str_cli (int sockfd)
{
    int n, i;
    char sendline[MAXLINE], recvline[MAXLINE + 1];
    while(1)
    {
        memset (sendline, 2, 128);
        if(written(sockfd, sendline, 126)!=126){
            printf("strcli:written error on sock\n");
        }
        i = readline(sockfd, recvline, 126);
    }
}
int readline(int fd, char *ptr, int maxlen)
{
    int n, rc;
    char c;
    for(n = 0; n < maxlen; n++){
        if((rc = read(fd, &c, 1)) == 1){
            *ptr++ = c;
            if(c=='\n'){
                break;
            }
            else if(rc== 0) {
                if(n== 0) {
                    return (0);
                }
            }
        }
    }
}

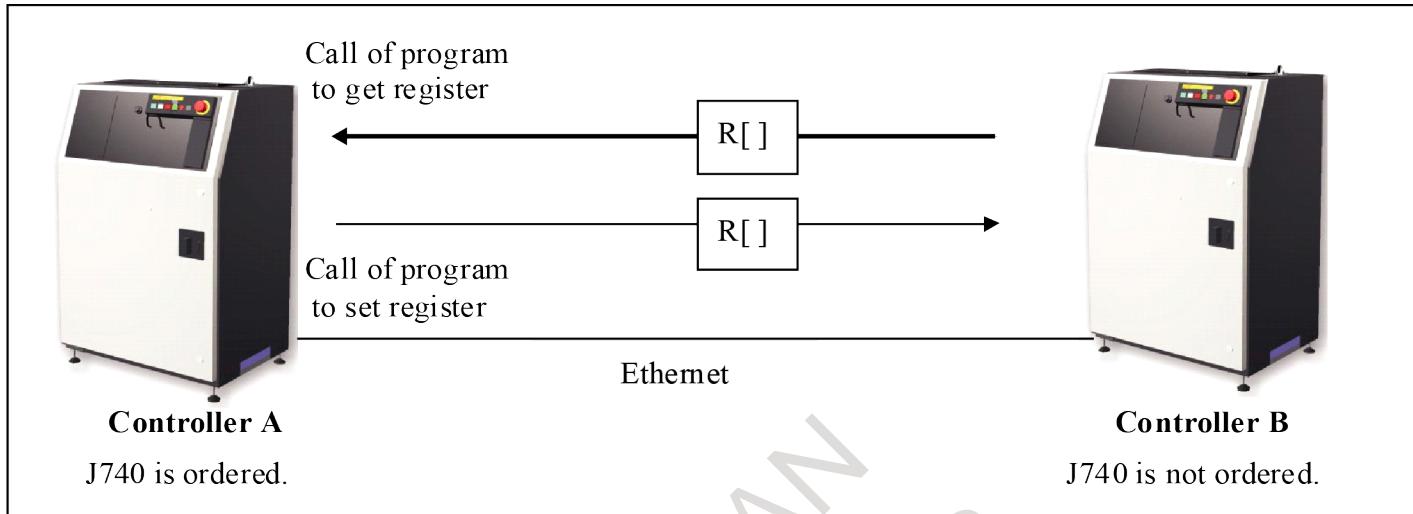
```

```
        else{
            break;
        }
    }
else{
    return (-1);
}
}
*ptr = 0;
return (n);
}
int written(int fd, char *ptr, int nbytes)
{
    int nleft, nwritten;
    nleft = nbytes;
    while(nleft > 0) {
        nwritten = write(fd, ptr, nleft);
        if(nwritten <= 0) {
            return(nwritten);
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(nbytes - nleft);
}
```

**Figure 12.5.3 ANSI C UNIX-Based Loopback Client Example**

## 13 DATA TRANSFER BETWEEN ROBOTS OVER ETHERNET (DTBR)

This section explains the optional (J740) data transfer between robots function available in V7.70 or later. This function enables you to transfer data between robots over Ethernet. By calling a KAREL program, you can transfer registers and position registers between robot controllers.



**Figure 13 Data Transfer Between Robots over Ethernet**

In [Figure 13](#), controller A runs the program to get the register from controller B. In addition, Controller A calls the program to transfer the register to controller B. In [Figure 13](#), only controller A starts data transfer. Controller B is just responding to request from controller A. In this case, only controller A needs this option.

## 13.1 TERMINOLOGY

To simplify and clarify explanation of this function we use following terms.

### Client

A client is a robot controller that starts communication by this function. In [Figure 13](#), controller A is the client. The client runs the program to request various services from another controller.

### Server

A Server is a robot controller that receives requests from a client and responds to each request. In [Figure 13](#), controller B is the server.

## 13.2 SETUP

Connect robot controllers by Ethernet. Set the TCP/IP parameters so that controllers can communicate over Ethernet by TCP/IP.

## 13.2.1 TCP/IP Setup

### Procedure

1. Press **MENU**.
2. Select **SETUP**.
3. Select **Host Comm.**.
4. Input host name (Robot name) and IP address and those of controllers with which to communicate. See [Figure 13.2.1](#).
5. Cycle controller power.

#### NOTE

Refer to Setting up TCP/IP in the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for more information.

#### NOTE

Do not use an underscore (\_) in the host name.

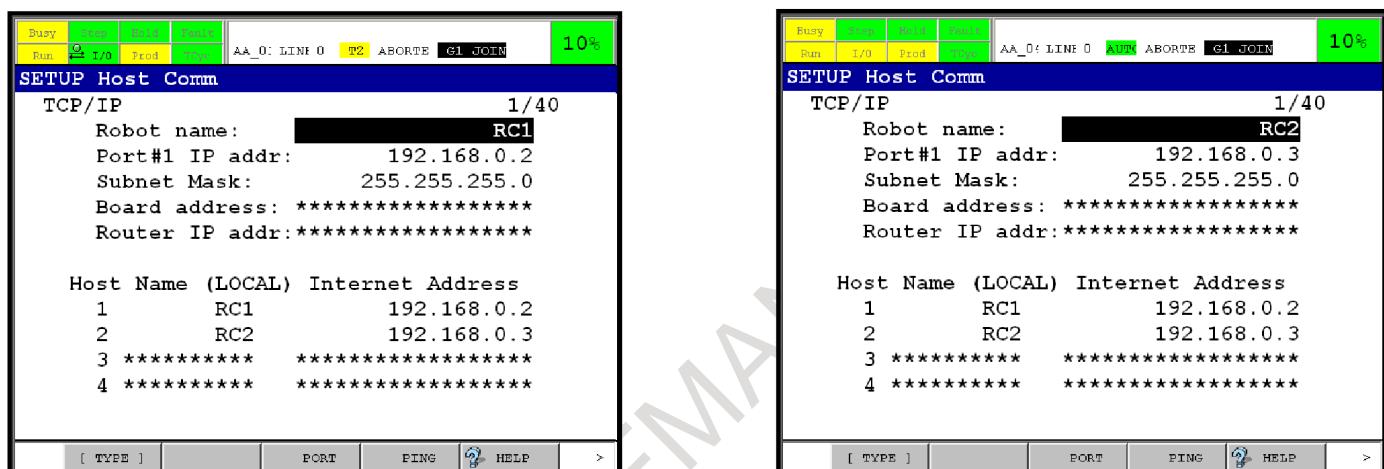
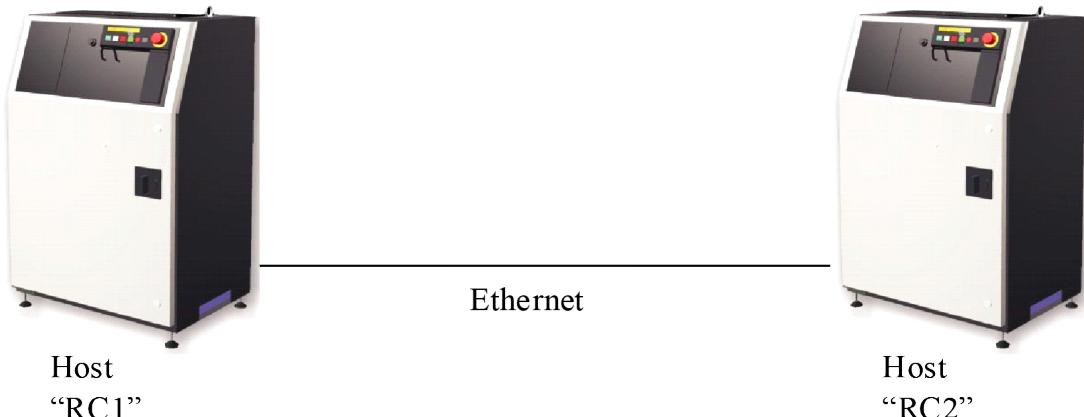


Figure 13.2.1 Setup Over Ethernet

### 13.3 TCP/IP SETUP FOR ROBOGUIDE

You can simulate data transfer in a work cell of ROBOGUIDE. Please note that the IP address of each host must be 127.0.0.x (where  $x \geq 2$ ). [Figure 13.3](#) is an example of TCIP/IP setup for ROBOGUIDE. For example, if there are two robot controllers in your workcell, the host name of the first one is RC1 and the second one is RC2.

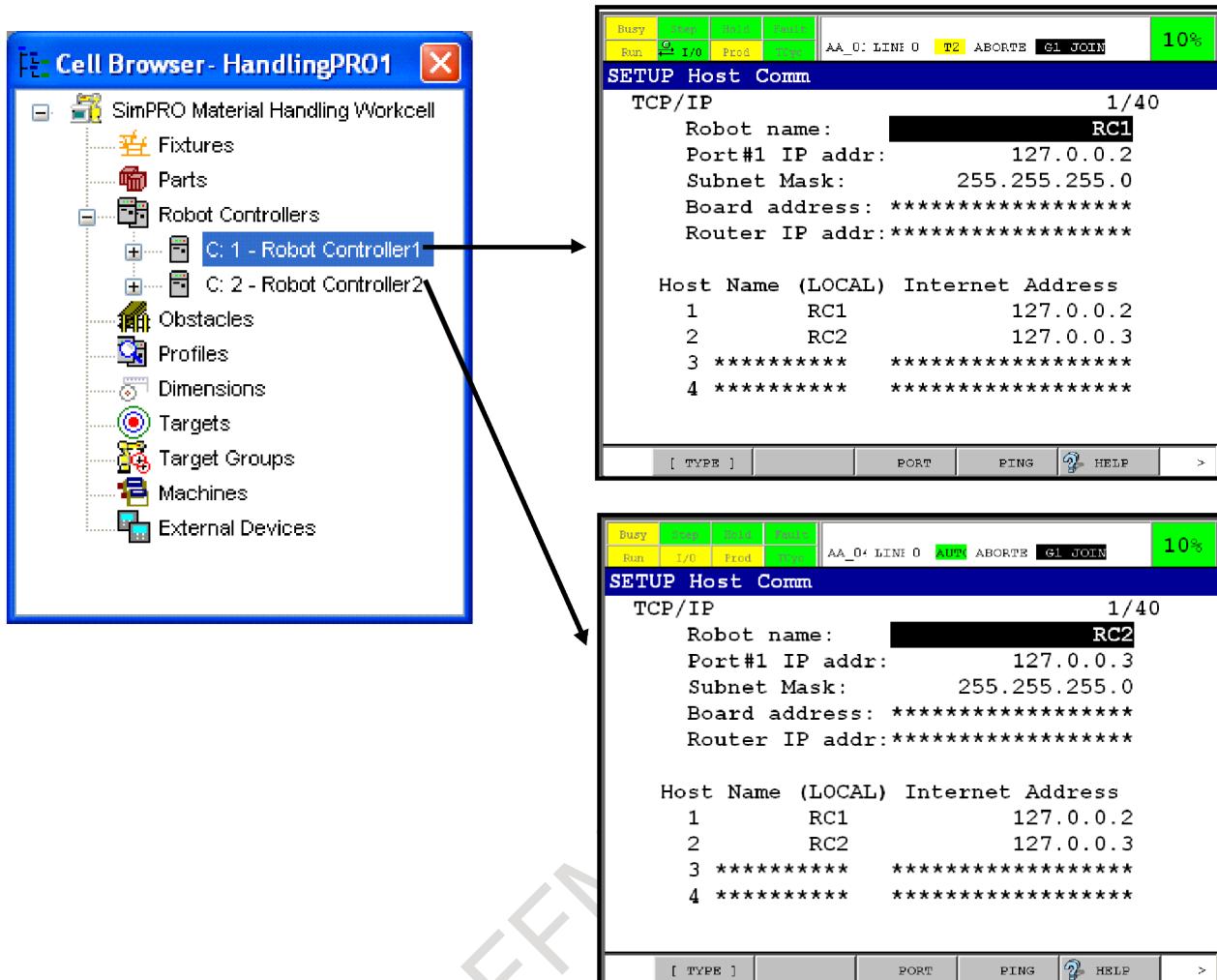


Figure 13.3 TCP/IP Setup for ROBOGUIDE

You have to re-start the robot controllers after TCP/IP setup. If you run the following program on RC2, the value and comment of R[1] of RC1 is transferred to R[5] of RC2.

```
1: CALL RGETNREG('RC1',1,5,0) ;
```

RGETNREG is a KAREL program this function provides. Refer to [Section 13.4.1, RGETNREG: Program to Get Numeric Register](#) for more information.

## 13.4 STANDARD DATA TRANSFER PROGRAMS

This function includes programs to transfer registers and position registers. See [Table 13.4](#).

**Table 13.4 Program List to Transfer Registers and Position Registers**

Program	Function	Syntax
RGETNREG	To get numeric register	RGETNREG( 'host/IP address', source index, destination index, option)
RSETNREG	To set numeric register	RSETNREG( 'host/IP address', destination index, source index, option)

Program	Function	Syntax
RGETPREG	To get position register	RGETPREG( 'host/IP address', source index, source group number, destination index, destination group, option)
RSETPREG	To set position register	RSETPREG( 'host/IP address', destination index, destination group, source index, source group number, option)

**NOTE**

If the arguments are not correct, an error will be displayed. The following are examples of incorrect arguments:

- Type of argument (integer, string and so forth) is wrong.
- The number of arguments is less than expected.
- The content of the argument is incorrect. Call the program with the proper arguments. When you restart execution of the program, move the cursor to 1 line before CALL, and restart from the line.

### 13.4.1 RGETNREG: Program to Get Numeric Register

**Program Name:** RGETNREG

**Overview:** The client controller gets the register from the server and sets it to the register of the client.

**Syntax:** RGETNREG (host\_name, src\_idx, dest\_idx, option)

Input/Output parameters:

The 1st parameter: [IN] String of host name or IP address of server.

The 2nd parameter: [IN] Index of source register. This integer is the index of the register of the server.

The 3rd parameter: [IN] Index of destination register. This integer is the index of the register of the client.

The 4th parameter: [IN] This integer specifies the function of this program

**Table 13.4.1 Parameter Values**

Value	Description
0	Value and comment are received.
1	Value is received.
2	Comment is received.

**Details:**

If R [src\_idx] is not on the server, **DTBR-014** Bad variable or register index (host name, index) is posted.

If R [dest\_idx] is not on the client, **VARS-024** Bad variable or register index is posted.

**Example:**

Execution of the following program gets the value and comment of R[10] of the Host ' SERVER '.

```
CALL RGETNREG (
  'SERVER', 10, 20, 0)
```

## 13.4.2 RSETNREG: Program to Set Numeric Register

**Program Name:** RSETNREG

**Overview:** Client controller sets the server's register from the client's register.

**Syntax:** RSETNREG (host\_name, dest\_idx, src\_idx, option)

Input/Output parameters:

The 1st parameter: [IN] String of host name or IP address of server.

The 2nd parameter: [IN] Index of destination register. This integer is the index of the register of the server.

The 3rd parameter: [IN] Index of the source register. This integer is the index of the register of the client.

The 4th parameter: [IN] This integer specifies the function of this program

**Table 13.4.2 Parameter Values**

Value	Description
0	Value and comment are set.
1	Value is set.

**Details:**

If R [dest\_idx] is not on the server, **DTBR-014** Bad variable or register index (host name, index) is posted.

If R [src\_idx] is not on the client, **VARS-024** Bad variable or register index is posted.

**Execution of the following program sets the value and comment of R [20] of the client to R [10] of the host ' SERVER '.**

```
CALL RSETNREG (
  'SERVER', 10, 20, 0)
```

## 13.4.3 RGETPREG: Program to Get Position Register

**Program Name:** RGETPREG

**Overview:** The client controller gets the position register from the server and sets it to the position register of the client. The position data of the specified group only is received.

**Syntax:** RGETPREG (host\_name, src\_idx, src\_grp, dest\_idx, dest\_grp, option)

Input/Output parameters:

The 1st parameter: [IN] String of host name or IP address of server.

The 2nd parameter: [IN] Index of source position register. This integer is the index of the register of the server.

The 3rd parameter: [IN] Group number of source position register. This integer is the group number of the server.

The 4th parameter: [IN] Index of destination register. This integer is the index of the register of the client.

The 5th parameter: [IN] Group number of the destination position register. This integer is the group number of the client.

The 6th parameter: [IN] This integer specifies the function of this program

**Table 13.4.3 (a) Parameter Values**

Value	Description
0	Value and comment are received.
1	Value is received.

#### Details:

If contents of the 2nd through the 5th parameters are incorrect, the following errors are posted.

**Table 13.4.3 (b) Possible Errors**

Error	Cause
<b>DTBR-014</b> Bad variable or register index (host name, index)	<b>PR [(the 2nd parameter) src_idx]</b> does not exist on the server.
<b>DTBR-15</b> Illegal group number (host name, index)	<b>Group (the 3rd parameter) src_grp</b> does not exist on the server.
<b>VARS-024</b> Bad variable or register index	<b>PR [(the 4th parameter) dest_idx ]</b> does not exist on the client.
<b>ROUT-026</b> Illegal group number	<b>Group (the 5th parameter)dest_grp</b> does not exist on the client.

#### NOTE

Position data of the specified groups of client and server may be different in its components. For example, the following points may differ:

- The number of axes
- Configuration of Cartesian position
- Type of axis (rotary or linear)

You may not be able to use position register that is received or set as it is. Please be careful to use transferred data.

#### NOTE

Even if Cartesian position data is received and stored to client's group that does not support Cartesian data (independent axis for example), Cartesian data is set.

#### Example:

Execution of following program by client transfers position data of group 2 of PR [10] of host 'SERVER' to client's group 1 of PR[20]. Comment is not acquired.

```
CALL RGETPREG (
  'SERVER', 10, 2, 20, 1, 1)
```

## 13.4.4 RSETPREG: Program to Set Position Register

### Program Name: RSETPREG

**Overview:** Client controller sets the server's position register from the client's position register. The position data of the specified group only is set.

**Syntax:** RSETPREG (host\_name, dest\_idx, dest\_grp, src\_idx, src\_grp, option)

Input/Output parameters:

The 1st parameter: [IN] String of host name or IP address of server.

The 2nd parameter: [IN] Index of destination position register. This integer is index of PR of server.

The 3rd parameter: [IN] Group number of position data of destination PR of server. This is INTEGER.

The 4th parameter: [IN] Index of source position register. This integer is index of PR of client.

The 5th parameter: [IN] Group number of position data of source PR of client. This is INTEGER

The 6th parameter: [IN] This integer specifies function of this program

**Table 13.4.4 (a) Parameter Values**

Value	Description
0	Position data and comment is set.
1	Position data is set.

### Details:

If contents of the 2nd through 5th parameters are wrong, the following errors are posted.

**Table 13.4.4 (b) Possible Errors**

Error	Cause
<b>DTBR-014</b> Bad variable or register index(host name, index)	<b>PR [(the 2nd parameter) dest_idx]</b> does not exist on the server.
<b>DTBR-15</b> Illegal group number(host name, index)	<b>Group (the 3rd parameter) dest_grp</b> does not exist on the server.
<b>VARS-024</b> Bad variable or register index	<b>PR [(the 4th parameter) dest_idx]</b> does not exist on the client.
<b>ROUT-026</b> Illegal group number	<b>Group (the 5th parameter) dest_grp</b> does not exist on the client.

**NOTE**

Position data of the specified groups of client and server may be different in its components. For example, the following points may differ.

- The number of axes
- Configuration of Cartesian position
- Type of axis (rotary or linear)

You may not be able to use position registers that is received or set as it is. Please be careful to use transferred data.

**NOTE**

Even if Cartesian position data is set to server's group that does not support Cartesian data (independent axis for example), Cartesian data is set.

**Example:**

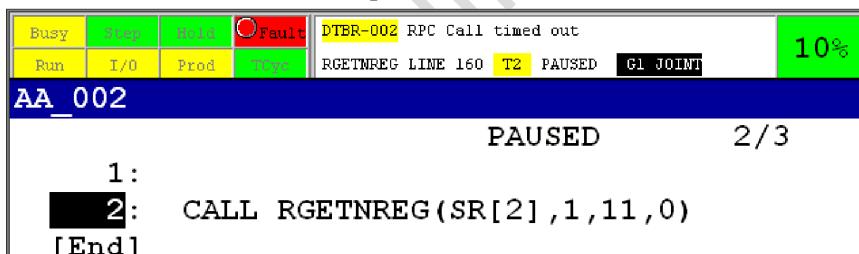
Execution of following program by client transfers position data and comment of group 1 of PR [20] of client to group 2 of PR [10] of host 'SERVER'.

```
CALL RSETPREG (
  'SERVER', 10, 2, 20, 1, 0)
```

## 13.5 ERROR RECOVERY

Most errors that occur at the execution of the standard data transfer program are of the following two cases:

1. **DTBR-002** RPC Call timed out is posted because communication failed or was not in time.



**Figure 13.5 (a) Error Recovery**

Check whether the controller (client) can always, reliably, communicate with the specified host (server). You can perform PING in PING screen of Host Comm setup screen. Improve network traffic. If the standard data transfer program or built-in of this function is used very often, lessen the frequency of use. If the standard data transfer program or built-in of this function is executed consecutively, there should be interval. If there is already interval, please lengthen it.

2. Arguments were wrong. In this case, arguments must be modified. In program edit screen, cursor is on the call of standard data transfer program. However, execution stopped inside of called program. Arguments were already handed to sub program (in figure below, RGETNREG). Even if you modify arguments in program edit screen, resume from the same line does not solve the problem. It is because value of arguments was already handed to sub program. If arguments are register or string register, value was handed at call. Even if you change value of register or string register after error happened,

the change is not reflected to resume from the same line. After modification of arguments, please move cursor to 1 line before CALL and resume from the line.

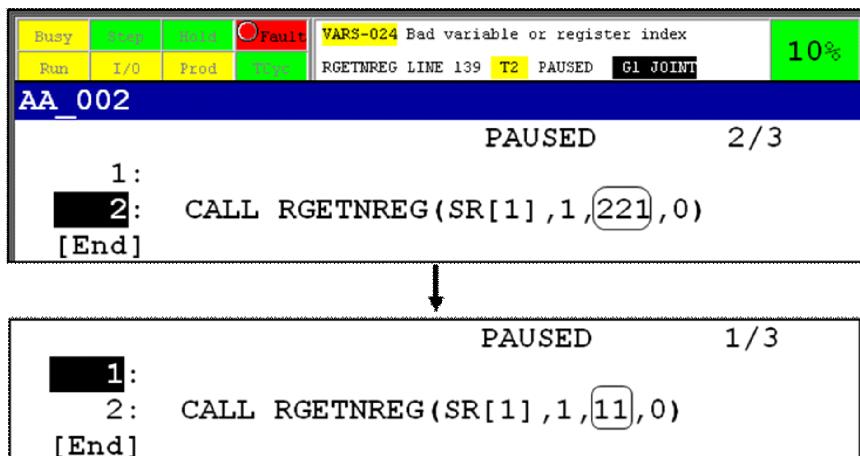


Figure 13.5 (b) Error Recovery 2

## 13.6 KAREL BUILT-INS

A KAREL built-in routine (or function) is a pre-defined function, which is built in to the KAREL language. This function provides built-in routines to transfer data between robots. The KAREL program using the built-in enables following:

- Sending and receiving register
- Sending and receiving position register
- Sending and receiving string register
- Read and write of I/O value
- Read of I/O simulation status
- Simulating and un-simulating I/O
- Read and write of comment of following data
  - Register
  - Position register
  - String register
  - I/O Following table lists built-in routines provided by this function.

Refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more information on the following built-ins.

Table 13.6 DTBR KAREL Built-In Routines

Built-In	Description
RGET_PORTCMT	Gets I/O comment
RGET_PORTSIM	Gets I/O simulation status
RGET_PORTVAL	Gets I/O value
RGET_PREGCMT	Gets comment of position register
RGET_REG	Gets register
RGET_REG_CMT	Gets comment of register
RGET_SREGCMT	Gets comment of string register

Built-In	Description
RGET_STR_REG	Gets string register
RNREG_RECV	Transfers server's register to client's register
RNREG_SEND	Transfers client's register to server's register
RPREG_RECV	Transfers server's position register to client's position register.
RPREG_SEND	Transfers client's position register to server's position register.
RSET_INT_REG	Sets INTEGER to register
RSET_PORTCMT	Sets I/O comment
RSET_PORTSIM	Simulates I/O
RSET_PORTVAL	Sets I/O value
RSET_PREGCMT	Sets comment of position register
RSET_REALREG	Sets REAL value to register
RSET_REG_CMT	Sets comment of register
RSET_SREGCMT	Sets comment of string register
RSET_STR_REG	Sets string register

## 13.7 TIME OUT AND RETRY

---

When communication does not complete in time because of a problem like network problems, this function disconnects the connection and tries to communicate again. After retrying two times, this function gives up communication. The standard data transfer program automatically posts **DTBR-002** RPC Call timed out. The status of the built-in of this function will be set to corresponding value. Timeout value is two seconds by default. The number of retries is two.

When communication completely fails, this function gives up in about six seconds. The time out value is stored in **\$DTBR\_CFG.\$RPC\_TIMEOUT**, in seconds. The minimum timeout value is one second regardless of the value of the system variable. If the controller cannot recognize the host name at all, communication can be given up before the time out happens. Detection of a communication error takes more time than usual if the timeout value is longer than the default. This function does not give up communication and retry even if the program is paused or aborted. If a timeout time is longer than usual, the time you have to wait before starting next communication becomes longer.

## 13.8 LIMITATIONS

---

- Performance of communication is not guaranteed. It depends network traffic and the load of the CPU of the controller.
- This function affects the performance of other functions that use Ethernet, and vice versa.
- Request from multiple tasks (programs) doesn't cause parallel execution of data transfer. They are actually processed sequentially.
- The system software version of the controller to communicate with (server) must be 7DA5 or later.
- You cannot restrict data transfer by the other functions, for example password function.
- A robot controller cannot send request to itself by this function.

- Transfer of position register pays attention to the number of axis or extended axis so that data to be set to destination does not exceed axes destination controller has. However, the other elements of position data are not considered, such as:
  - Whether the robot can have Cartesian data or not.
  - Axis limit
  - Configuration of position available and range of turn number.
  - Type of axis (rotary or linear) H) Pause or force abort of program does not stop communication once it is started. Next request should be issued after completion of the previous one.

 **CAUTION**

1. This function can not be used instead of hardwired I/O, field network. This function offers the ability to read/write I/O value of remote controller. However, hardwired I/O and field network I/O is always faster.
2. Cyclic execution increases load of network and CPU of controller. Consecutive or cyclic execution of data transfer should be avoided
3. Caution of transfer of position register Content of position register depends on type of robots and configuration of position. Please be careful when using transferred position registers. Especially, you should be very careful when you use transferred position registers as a destination point of the motion statement Server and client robots are not always robots of the same type. Interpretation of position registers depends on the currently selected tool and user frame.

When you transfer a register or position register, please note that only one controller changes a register or position register. This is also true of string registers and I/O if you handle them using built-ins.

If both server and client write to the same data, they overwrite the value written by the other controller. This might look as if data transfer was not working properly. Especially, if timing of the write by the client and server is very close, the earlier one may not be recognizable.

When the client writes a register, position register or string register of server, value and comment is acquired first. Then the client overwrites the necessary part of the acquired data and request server to write the updated value as a whole. If the server writes slightly before the client, the value or comment might be changed back to those that were received by the client.

## 13.9 TROUBLESHOOTING

### DTBR-002 RPC Call timed out is posted

Check if the controller (client) can always, stably communicate with the specified host (server). You can perform PING in PING screen of **Host Comm** setup screen. If the standard data transfer program or built-in of this function is used very often, please lessen the frequency of use. If the standard data transfer program or built-in of this function is executed consecutively, there should be an interval. If there is already interval, please lengthen it.

### Modification of arguments of program is not reflected after resume

If you modified arguments of the standard data transfer program, please move the cursor to one line before the CALL and resume the program from the line.

**Communication takes time or speed is unstable.**

Speed of data transfer by this function is not guaranteed. Performance depends on network traffic, load of CPU of client and server. Please improve network traffic. Please improve timing of data transfer. For example, data transfer that is not needed during motion of robots can be done when robot is not moving.

**INTP-320 (program name, line number) Undefined built-in is posted.**

If this symptom happens when standard data transfer program or built-in of this function is executed, please confirm whether this function is ordered or not. This function is optional.

**Time out does not happen. Program keeps running.**

Pause or abort of program during data transfer does not stop the transfer once started. It tries to transfer data until the last retry fails. The next request of data transfer is not processed until previous data transfer completes (fails). See the following procedure as an example.

1. Ethernet cable is not plugged in properly.
2. Standard data transfer program, RGETNREG is called but user aborted the program immediately by FCTN menu.
3. Run the same program again and RGETNREG is called immediately.

**Time out error happens just after program resume.**

Program pause does not stop data transfer once it started. If a timeout occurs while the program is paused, a timeout error is posted at resume.

**Value of axis of joint position of PR[ ] is not transferred properly.**

If type of axis (rotary or linear) is different between client's and server's corresponding axes, transferred value is not interpreted in the same way. It is not treated as the same value.

**INTP-311 (program name, line number) Uninitialized data is used is posted when transferred position register is used.**

1. Please confirm source data is initialized.
2. The number of total axes or extended axes might be different. If the source data does not have enough axes to set all axis data to destination data, axes that cannot be set because of lack of data becomes uninitialized. In this case, proper data should be set before use. Suppose group 1 of host RC1 has an extended axis and group 1 of RC2 has no extended axes. If Cartesian position of group 1 of RC2 is set to group 1 of RC1, E1 is set to an un-initialized value. RC1 have to set proper value by it self. If 0 is appropriate, PR[1,7]=0 can be used.

**Position data of sent/received position register is partially un-initialized.**

Please refer to the previous item. The position register screen does not display R unless data of all groups are not initialized. Standard transfer program and built-in transfers only specified group.

**VARS-037 Position register is locked is posted.**

Please check if position register is locked on server or client.

**HRTL-047 Address family not supported is posted.**

Please check if the specified host is the client itself. The client cannot operate data of client itself by this function.

**Data is sent to/received from a controller different from specified one.**

If you did not cycle power after changing the IP address and host name, please cycle power. If host name includes an underscore, do not use it.

**HRTL-049 Can't assign requested address is posted.**

**HOST-108 Internet address not found is posted.**

Please check if the specified host name or IP address is correct. If you did not cycle power after changing the IP address and host name, please cycle power. Please check if the controller (client) can always, stably communicate with the specified host (server). You can perform PING in PING screen of **Host Comm setup** screen.

EFFMAN  
QUIRIONR

# 14 SYSTEM VARIABLES

---

System variables are variables that are declared as part of the KAREL system software. They have permanently defined variable names that begin with a dollar sign (\$). Many system variables are *structure variables*, in which case each field also begins with a dollar sign (\$). Many are robot specific, meaning their values depend on the type of robot that is attached to the system.

System variables have the following characteristics:

- They have predefined data types that can be any one of the valid KAREL data types.
- The initial values of the system variables are either internal default values or variables stored in the default system variable file, SYSDEF.SV.
- When loading and saving system variables from the FILE screen or KCL, the system variable file name defaults to SYSVARS.SV.
- Access rights govern whether or not you can examine or change system variables.
- Modified system variables can be saved to reflect the current status of the system.

**See Also:** [Chapter 2, LANGUAGE ELEMENTS](#) for more information on the data types available in KAREL

## 14.1 ACCESS RIGHTS

---

The following rules apply to system variables:

- If a system variable allows a KAREL program to read its value, you can use that value in the same context as you use program variable values or constant values in KAREL programs.

For example, these system variables can be used on the right hand side of an assignment statement or as a test condition in a control statement.

- If a system variable allows a KAREL program to write its value, you can use that system variable in any context where you assign values to variables in KAREL programs.

The symbols for the program access rights are listed in [Table 14.1](#).

**Table 14.1 Access Rights for System Variables**

Access	Meaning
NO	No access
RO	Read only
RW	Read and write
FP	Field protection; if it is a structure variable, one of the first three access rights will apply.

## 14.2 STORAGE

---

System variables are assigned an initial value upon power up based on

- Internal default values
- Values stored in the default system variable file, SYSDEF.SV

EFFMAN  
QUIRIONR

# 15 KAREL COMMAND LANGUAGE (KCL)

The KAREL command language (KCL) environment contains a group of commands that can be used to direct the KAREL system. KCL commands allow you to develop and execute programs, work with files, get information about the system, and perform many other daily operations.

The KCL environment can be displayed on the CRT/KB by pressing **MENUS** (F10) and selecting **KCL** from the menu.

In addition to entering commands directly at the KCL prompt, KCL commands can be executed from command files.

## 15.1 COMMAND FORMAT

A command entry consists of the command keyword and any arguments or parameters that are associated with that command. Some commands also require identifiers specifying the object of the command.

- KCL command keywords are action words such as **LOAD**, **EDIT**, and **RUN**. Command arguments, or parameters, help to define on what object the keyword is supposed to act.
- Many KCL commands have default arguments associated with them. For these commands, you need to enter only the keyword and the system will supply the default arguments.
- KCL supports the use of an asterisk (\*) as a wildcard, which allows you to specify a group of objects as a command argument for the following KCL commands:
  - **COPY**
  - **DELETE FILE**
  - **DIRECTORY**
- KCL identifiers follow the same rules as the identifiers in the KAREL programming language.
- All of the data types supported by the KAREL programming language are supported in KCL. Therefore, you can create and set variables in KCL.

**See Also:** [Chapter 2, LANGUAGE ELEMENTS](#) , and [Chapter 10, FILE SYSTEM](#)

### 15.1.1 Default Program

Setting a program name as a default for program name arguments and file name arguments allows you to issue a KCL command without typing the name.

The KCL default program can be set by doing one of the following:

- Using the **SET DEFAULT** KCL command
- Selecting a program name at the **SELECT** menu on the CRT/KB

### 15.1.2 Variables and Data Types

The **KCL> CREATE VARIABLE** command allows you to declare variables. The **KCL> SET VARIABLE** command permits you to assign values to declared variables. Assigned values can be

INTEGER, REAL, BOOLEAN, and STRING data types. Values can be assigned to particular ARRAY elements or specified PATH nodes. VECTOR variables are assigned as three REAL values, and POSITION variables are assigned as six REAL values.

**See Also:** [Section C.15, CREATE VARIABLE command](#) and [Section C.69, SET VARIABLE command](#) in [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#).

## 15.2 PROGRAM CONTROL COMMANDS

KCL commands can be used to run programs. In some cases, these programs may cause motion such as when a teach pendant program is run or when a KAREL program that calls a teach pendant program is run. The device from which the KCL command is issued must have motion control in order to do this.

Program control commands:

- Can immediately cause robot and/or auxiliary axis motion, or have the potential to cause motion
- Can be executed only if a number of conditions are met

The following commands are program control commands:

- CONTINUE
- RUN

 **WARNING**

Be sure that the robot work envelope is clear of personnel before issuing a program control command or starting a robot that automatically executes a program at power up. Otherwise, you could injure personnel or damage equipment.

## 15.3 ENTERING COMMANDS

You can enter KCL commands only from the CRT/KB.

To enter KCL commands:

1. Press **MENU (F10)** at the CRT/KB.
2. Select **KCL**.
3. Enter commands at the KCL prompt.

By entering the first keyword of a KCL command that requires more than one keyword, and by pressing ENTER, a list of all additional KCL keywords will be displayed.

For example, entering **DELETE** at the KCL prompt will display the following list of possible commands: FILE, NODE, or VARIABLE.

**NOTE**

The up arrow key can be used to recall any of the last ten commands entered.

### 15.3.1 Abbreviations

Any KCL command can be abbreviated as long as the abbreviations are unique in KCL. For example, TRAN is unique to TRANSLATE and ED, to EDIT.

### 15.3.2 Error Messages

If you enter a KCL command incorrectly, KCL displays the appropriate error message and returns the KCL> prompt, allowing you to reenter the command. An up arrow (^) indicates the offending character or the beginning of the offending word.

### 15.3.3 Subdirectories

Subdirectories are available on the memory card device. Subdirectories allow both memory cards and Flash disk cards to be formatted on any MS-DOS file system. You can perform all KCL file related commands on subdirectories. You can nest subdirectories up to many levels. However, FANUC does not recommend nesting subdirectories greater than eight levels.

## 15.4 COMMAND PROCEDURES

Command procedures are a sequence of KCL commands that are stored in a command file (.CF file type) and can be executed automatically in sequence.

- Command procedures allow you to use a sequence of KCL commands without typing them over and over.
- Command procedures are executed using the RUNCF command.

### 15.4.1 Command Procedure Format

All KCL commands except RUNCF can be used inside a command procedure. For commands that require confirmation, you can enter either the command and confirmation on one line or KCL will prompt for the confirmation on the input line. [Figure 15.4.1](#) displays **CLEAR ALL** as the KCL command and **YES** as the confirmation.

Enter command and confirmation on one line:  
CLEAR ALL YES

**Figure 15.4.1 Confirmation in a Command Procedure**

#### Nesting Command Procedures

Use the following guidelines when nesting command procedures:

- Command procedures can be nested by using %INCLUDE filename inside a command procedure.
- Nesting of command procedures is restricted to four levels.

If nesting of more than four command procedures is attempted, KCL will detect the error and take the appropriate action based on the system variable `$STOP_ON_ERR`. Refer to [Section 15.4.3, Error Processing](#) for more information on `$STOP_ON_ERR`.

**See Also:** [Section 15.4.3, Error Processing](#)

### Continuation Character

The KCL continuation character, ampersand (&), allows you to continue a command entry across more than one line in a command procedure.

You can break up KCL commands between keywords or between special characters.

For example, use the ampersand (&) to continue a command across two lines:

```
CREATE VAR [ TESTING_PROG.] PICK_UP_PNT &
:POSITION
```

### Comments

Comment lines can be used to document command procedures. The following rules apply to using comments in command procedures:

- Precede comments with two consecutive hyphens (--).
- Comments can be placed on a line by themselves or at the end of a command line.

## 15.4.2 Creating Command Procedures

A command procedure can be created by typing in the list of commands into a command file and saving the file. This can be done using the full screen editor.

**See Also:** EDIT KCL commands, [Chapter 15, KAREL COMMAND LANGUAGE \(KCL\)](#).

## 15.4.3 Error Processing

If the system detects a KCL error while a command procedure is being executed, the system handles the error in one of two ways, depending on the value of the system variable `$STOP_ON_ERR`:

- If `$STOP_ON_ERR` is TRUE when a KCL error is detected, the command procedure terminates and the KCL> prompt returns.
- If `$STOP_ON_ERR` is FALSE, the system ignores KCL errors and the command procedure runs to completion.

## 15.4.4 Executing Command Procedures

Each command in a command procedure is displayed as it is executed unless the SET VERIFY OFF command is used. Each command is preceded with the line number from the command file. However, if the file is not on the RD: device, the entire command file is read into memory before execution and line numbers will be omitted from the display.

Command procedures can be executed using the KCL RUNCF command.

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# 16 INPUT/OUTPUT SYSTEM

---

The Input/Output (I/O) system provides user access with KAREL to user-defined I/O signals, system-defined I/O signals and communication ports. The user-defined I/O signals are controlled in a KAREL program and allow you to communicate with peripheral devices and the robot end-of-arm tooling. System-defined I/O signals are those that are designated by the KAREL system for specific purposes. Standard and optional communications port configurations also exist.

The number of user-defined I/O signals is dependent on the controller hardware and on the types and number of modules selected.

## 16.1 USER-DEFINED SIGNALS

---

*User-defined signals* are those input and output signals whose meaning is defined by a KAREL program. You have access to user-defined signals through the following predefined port arrays:

- DIN (digital input) and DOUT (digital output)
- GIN (group input) and GOUT (group output)
- AIN (analog input) and AOUT (analog output)

In addition to the port arrays, you have access to robot hand control signals through KAREL OPEN and CLOSE HAND statements.

### 16.1.1 DIN and DOUT Signals

---

The DIN and DOUT signals provide access to data on a single input or output line in a KAREL program.

The program treats the data as a BOOLEAN data type. The value is either ON (active) or OFF (inactive). You can define the polarity of the signal as either active-high (ON when voltage is applied) or active-low (ON when voltage is not applied).

Input signals are accessed in a KAREL program by the name `DIN[n]`, where `n` is the signal number.

Evaluating DIN signals causes the system to perform read operations of the input port. Assigning a value to a DIN signal is an invalid operation unless the DIN signal has been simulated. These can never be set in a KAREL program, unless the DIN signal has been simulated.

Evaluating DOUT signals causes the system to return the currently output value from the specified output signal. Assigning a value to a DOUT signal causes the system to set the output signal to ON or OFF.

#### To turn on a DOUT:

```
DOUT[n] = TRUE or  
DOUT[n] = ON
```

#### To turn off a DOUT:

```
DOUT[n] = FALSE or  
DOUT[n] = OFF
```

You assign digital signals to the ports on I/O devices using teach pendant I/O menus or the KAREL built-in routine `SET_PORT_ASG`.

## 16.1.2 GIN and GOUT Signals

---

The GIN and GOUT signals provide access to DINs and DOUTs as a group of input or output signals in a KAREL program. A group can have a size of 1 to 16 bits, with each bit corresponding to an input or output signal. You define the group size and the DINs or DOUTs associated with a specific group. The first (lowest numbered) port is the least significant bit of the group value.

The program treats the data as an INTEGER data type. The unused bits are interpreted as zeros.

Input signals are accessed in KAREL programs by the name `GIN[n]`, where `n` is the group number.

Evaluating GIN signals causes the system to perform read operations of the input ports. Assigning a value to a GIN signal is an invalid operation unless the GIN signal has been simulated. These can never be set in a KAREL program, unless the GIN signal has been simulated.

Setting GOUT signals causes the system to return the currently output value from the specified output port. Assigning a value to a GOUT signal causes the system to perform an output operation.

To control a group output, the integer value equivalent to the desired binary output is used. For example the command `GOUT[n]=25` will have the following binary result `0000000000011001`, where 1 = output on and 0 = output off, least significant bit (LSB) being the first bit on the right.

You assign group signals using teach pendant I/O menus or the KAREL built-in routine `SET_PORT_ASG`.

## 16.1.3 AIN and AOUT Signals

---

The AIN and AOUT signals provide access to analog electrical signals in a KAREL program. For input signals, the analog data is digitized by the system and passed to the KAREL program as a 16 bit binary number, of which 14 bits, 12 bits, or 8 bits are significant depending on the analog module. The program treats the data as an INTEGER data type. For output signals, an analog voltage corresponding to a programmed INTEGER value is output.

Input signals are accessed in KAREL programs by the name `AIN[n]`, where `n` is the signal number.

Evaluating AIN signals causes the system to perform read operations of the input port. Setting an AIN signal at the teach pendant is an invalid operation unless the AIN signal has been simulated. These can never be set in a KAREL program, unless the AIN signal has been simulated.

The value displayed on the TP or read by a program from an analog input port are dependent on the voltage supplied to the port and the number of bits of significant data supplied by the analog-to-digital conversion. For positive input voltages, the values read will be in the range from 0 to  $2^{**}(N-1)-1$ , where `N` is the number of bits of significant data. For 12 bit devices (most FANUC modules), this is  $2^{**}11-1$ , or 2047.

For negative input voltages, the value will be in the range  $2^{**}N - 1$  to  $2^{**}(N-1)$  as the voltage varies from the smallest detectable negative voltage to the largest negative voltage handled by the device. For 12 bit devices, this is from 4095 to 2048.

An example of the KAREL logic for converting this input to a real value representing the voltage, where the device is a 12 bit device which handles a range from +10v to -10v would be as follows:

```
V: REAL
AINP: INTEGER
AINP = AIN[1]
IF (AINP <= 2047) THEN
V = AINP * 10.0 / 2047.0
ELSE
V = (AINP - 4096) * 10.0 / 2047
ENDIF
```

**Figure 16.1.3 KAREL Logic for Converting Input to a Real Value Representing the Voltage**

In TPP, the following logic would be used:

```
R[1] = AI[1]
IF (R[1] > 2047) JMP LBL[1]
R[2] = R[1] * 10
R[2] = R[2] / 2047
JMP LBL[2]
LBL[1]:
R[2] = R[1] - 4096
R[2] = R[2] * 10
R[2] = R[2] / 2047
LBL[2]
```

R[2] has the desired voltage.

Evaluating AOUT signals causes the system to return the currently output value from the specified output signal. Assigning a value to an AOUT signal causes the system to perform an output operation.

An AOUT can be turned on in a KAREL program with AOUT[n]=(an integer value). The result will be the output voltage on the AOUT signal line[n] of the integer value specified. For example, AOUT[1]=1000 will output a +5 V signal on Analog Output line 1 (using an output module with 12 significant bits).

You assign analog signals using teach pendant I/O menus or the KAREL built-in routine SET\_PORT\_ASG.

## 16.1.4 Hand Signals

You have access to a special set of robot hand control signals used to control end-of-arm tooling through the KAREL language HAND statements, rather than through port arrays. HAND signals provide a KAREL program with access to two output signals that work in a coordinated manner to control the tool. The signals are designated as the open line and the close line. The system can support up to two HAND signals.

HAND[1] uses the same physical outputs as RDO[1] and RDO[2].

HAND[2] uses the same physical outputs as RDO[3] and RDO[4].

The following KAREL language statements are provided for controlling the signal, where n is the signal number.

OPEN HAND n activates open line, and deactivates close line

CLOSE HAND n deactivates open line, and activates close line

RELAX HAND n deactivates both lines

## 16.2 SYSTEM-DEFINED SIGNALS

---

System-defined I/O signals are signals designated by the controller software for a specific purpose. Except for certain UOP signals, system-defined I/O cannot be reassigned.

You have access to system-defined I/O signals through the following port arrays:

- Robot digital input (RDI) and robot digital output (RDO)
- Operator panel input (OPIN) and operator panel output (OPOUT)
- Teach pendant input (TPIN) and teach pendant output (TPOUT)

### 16.2.1 Robot Digital Input and Output Signals (RDI/RDO)

---

Robot I/O is the input and output signals between the controller and the robot. These signals are sent to the EE (End Effector) connector located on the robot. The number of robot input and output signals (RDI and RDO) varies depending on the number of axes in the system. For more information on configuring Robot I/O, refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).

**RDI[1] through RDI[8]** are available for tool inputs. All or some of these signals can be used, depending on the robot model. Refer to the Maintenance Manual specific to your robot model, for more information.

**RDO[1] through RDO[8]** are available for tool control. All or some of these signals can be used, depending on the robot model. Refer to the Maintenance Manual specific to your robot model, for more information.

**RDO[1] through RDO[4]** are the same signals set using OPEN, CLOSE, and RELAX hand. See [Section 16.1.4, Hand Signals](#).

### 16.2.2 Operator Panel Input and Output Signals (OPIN/OPOUT)

---

Operator panel input and output signals are the input and output signals for the standard operator panel (SOP) and for the user operator panel (UOP).

Operator panel input signals are assigned as follows:

- The first 16 signals, OPIN[0] - OPIN[15], are assigned to the standard operator panel.
- The next 18 signals, OPIN[16] - OPIN[33], are assigned to the user operator panel (UOP). If you have a process I/O board, these 18 UOP signals are mapped to the first 18 input ports on the process I/O board.

Operator panel output signals are assigned as follows:

- The first 16 signals, OPOUT[0] - OPOUT[15], are assigned to the standard operator panel.

- The next 20 signals, OPOUT[16] - OPOUT[35], are assigned to the user operator panel (UOP). If you have a process I/O board, these 20 UOP signals are mapped to the first 20 output ports on the process I/O board.

### Standard Operator Panel Input and Output Signals

Standard operator panel input and output signals are recognized by the KAREL system as OPIN[0] - OPIN[15] and OPOUT[0] - OPOUT[15] and by the screens on the teach pendant as SI[0] - SI[15] and SO[0] - SO[15]. [Table 16.2.2 \(a\)](#) lists each standard operator panel input signal. [Table 16.2.2 \(b\)](#) lists each standard operator panel output signal.

**Table 16.2.2 (a) Standard Operator Panel Input Signals**

OPIN[n]	SI[n]	Function	Description
OPIN[0]	SI[0]	NOT USED	-
OPIN[1]	SI[1]	FAULT RESET	This signal is normally turned OFF, indicating that the FAULT RESET button is not being pressed.
OPIN[2]	SI[2]	REMOTE	This signal is normally turned OFF, indicating that the controller is not set to remote.
OPIN[3]	SI[3]	HOLD	This signal is normally turned ON, indicating that the HOLD button is not being pressed.
OPIN[6]	SI[6]	CYCLE START	This signal is normally turned OFF, indicating that the CYCLE START button is not being pressed.
OPIN[7] - OPIN[15]	SI[7], SI[10] - SI[15]	NOT USED	-
	SI[8] SI[9]	CE/CR Select b0 CE/CR Select b1	This signal is two bits and indicates the status of the mode select switch.

**Table 16.2.2 (b) Standard Operator Panel Output Signals**

OPOUT[n]	SOI[n]	Function	Description
OPOUT[0]	SO[0]	REMOTE LED	This signal indicates that the controller is set to remote.
OPOUT[1]	SO[1]	CYCLE START	This signal indicates that the CYCLE START button has been pressed or that a program is running.
OPOUT[2]	SO[2]	HOLD	This signal indicates that the HOLD button has been pressed or that a hold condition exists.
OPOUT[3]	SO[3]	FAULT LED	This signal indicates that a fault has occurred.
OPOUT[4]	SO[4]	BATTERY ALARM	This signal indicates that the CMOS battery voltage is low.
OPOUT[5]	SO[5]	USER LED#1 (PURGE COMPLETE for P-series robots)	This signal is user-definable.
OPOUT[6]	SO[6]	USER LED#2	This signal is user-definable.

OPOUT[n]	SOI[n]	Function	Description
OPOUT[7]	SO[7]	TEACH PENDANT ENABLED	This signal indicates that the teach pendant is enabled.
OPOUT[8] - OPOUT[15]	SO[8] - SO[15]	NOT USED	-

### User Operator Panel Input and Output Signals

User operator panel input and output signals are recognized by the KAREL system as OPIN[16]-OPIN[33] and OPOUT[16]-OPOUT[35] and by the screens on the teach pendant as UI[1]-UI[18] and UO[1]-UO[20]. On the process I/O board, UOP input signals are mapped to the first 18 digital input signals and UOP output signals are mapped to the first 20 digital output signals. [Table 16.2.2 \(c\)](#) lists and describes each user operator panel input signal. [Table 16.2.2 \(d\)](#) lists each user operator panel output signal. [Figure 16.2.2 \(a\)](#) and [Figure 16.2.2 \(b\)](#) illustrate the timing of the UOP signals.

**Table 16.2.2 (c) User Operator Panel Input Signals**

OPIN[n]	UOP Input Signal	Description
OPIN[16]	UI[1] *IMSTP Always active	<p>*IMSTP is the immediate stop software signal. *IMSTP is a normally OFF signal held ON. When it is set to OFF, it</p> <ul style="list-style-type: none"> <li>Pauses a program if one is running</li> <li>Immediately stops the robot and applies robot brakes</li> <li>Shuts off power to the servos</li> </ul> <p>Error code <b>SRVO-037</b> *IMSTP Input (Group:i) will be displayed when this signal is lost. This signal is always active.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><b>⚠ WARNING</b></p> <p>*IMSTP is a software controlled input and cannot be used for safety purposes. Use *IMSTP with EMG1, EMG2, and EMGCOM to use this signal with a hardware controller emergency stop. Refer to the Maintenance Manual specific to your robot model for connection information of EMG1, EMG2, and EMGCOM.</p> </div>
OPIN[17]	UI[2] *HOLD Always active	<p>*HOLD is the external hold signal. *HOLD is a normally OFF signal held ON. When it is set to OFF, it will do the following:</p> <ul style="list-style-type: none"> <li>Pause program execution.</li> <li>Slow motion to a controlled stop and hold.</li> <li>Optional Brake on Hold shuts off servo power after the robot stops.</li> </ul>
OPIN[18]	UI[3] *SFSPD Always active	<p>*SFSPD is the safety speed input signal. This signal is usually connected to the safety fence.*SFSPD is a normally OFF signal held ON, When it is set to OFF it will do the following:</p> <ul style="list-style-type: none"> <li>Pause program execution.</li> <li>Reduce the speed override value to that defined in a system variable. This value cannot be increased while *SFSPD is OFF.</li> <li>Display error code message <b>SYST-009</b>.</li> <li>Not allow a REMOTE start condition. Start inputs from UOP or SOP are disabled when *SFSPD is set to OFF and only the teach pendant has motion control with the speed clamped.</li> </ul>

OPIN[n]	UOP Input Signal	Description
OPIN[19]	UI[4] CSTOPI Always active	<p>CSTOPI is the cycle stop input. The function of this signal depends on the system variable <b>\$SHELL_CFG.\$USE_ABORT</b>.</p> <p>If the system variable <b>\$SHELL_CFG.\$USE_ABORT</b> is set to <b>FALSE</b>, the CSTOPI input</p> <ul style="list-style-type: none"> <li>Clears the queue of programs to be executed that were sent by RSR signals.</li> </ul> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><b>⚠ WARNING</b></p> <p>When <b>\$SHELL_CFG.USE_ABORT</b> is FALSE, CSTOPI does not immediately stop automatic program execution.</p> </div> <ul style="list-style-type: none"> <li>Automatic execution will be stopped after the current program has finished executing.</li> </ul> <p>If the system variable <b>\$SHELL_CFG.\$USE_ABORT</b> is set to <b>TRUE</b>, the CSTOPI input</p> <ul style="list-style-type: none"> <li>Clears the queue of programs to be executed that were sent by RSR signals.</li> <li>Immediately aborts the currently executing program for programs that were sent to be executed by either RSR or PNS.</li> </ul>
OPIN[20]	UI[5] FAULT_RESET Always active	FAULT_RESET is the external fault reset signal. When this signal is received the following will happen: <ul style="list-style-type: none"> <li>Error status is cleared.</li> <li>Servo power is turned ON.</li> <li>A paused program <b>will not be resumed</b>.</li> </ul>
OPIN[21]	UI[6] START Active when the robot is in a remote condition (CMDENBL = ON)	<p>START is the remote start input. How this signal functions depends on the system variable <b>\$SHELL_CFG.\$CONT_ONLY</b>.</p> <p>If the system variable <b>\$SHELL_CFG.\$CONT_ONLY</b> is set to <b>FALSE</b>, the START input signal will</p> <ul style="list-style-type: none"> <li>Resume a paused program.</li> <li>If a program is not paused, the currently selected program starts from the position of the cursor.</li> </ul> <p>If the system variable <b>\$SHELL_CFG.\$CONT_ONLY</b> is set to <b>TRUE</b>, the START input signal will</p> <ul style="list-style-type: none"> <li>Resume a paused program only. The PROD_START input must be used to start a program from the beginning.</li> </ul>
OPIN[22]	UI[7] HOME Active when the robot is in a remote condition	HOME is the home input. When this signal is received the robot moves to the defined home position. You configure the system to do this by setting up a macro program to run when UI[7] is received.
OPIN[23]	UI[8] ENBL Always active	ENBL is the enable input. This signal must be ON to have motion control ability. When this signal is OFF, robot motion cannot be done. When ENBL is ON and the System Configuration screen setting Remote/Local setup: is REMOTE, the robot is in a remote operating condition.

OPIN[n]	UOP Input Signal	Description
OPIN[24] - OPIN[27]	UI[9] - UI[12] RSR 1-4 Active when the robot is in a remote condition (CMDENBL = ON)	RSR1-4 are the robot service request input signals. When one of these signals is received, the corresponding RSR program is executed or, if a program is running currently, stored in a queue for later execution. RSR signals are used for production operation and can be received while an ACK output is being pulsed. See <a href="#">Figure 16.2.2 (a)</a> .
OPIN[24]-OPIN[31]	UI[9] - UI[16] PNS 1-8 Active when the robot is in a remote condition (CMDENBL = ON)	PNS 1-8 are program number select input signals. PNS selects programs for execution, <b>but does not execute programs</b> . Programs that are selected by PNS are executed using the START input or the PROD_START input depending on the value of the system variable \$SHELL_CFG.\$CONT_ONLY. The PNS number is output by pulsing the SNO signal (selected number output) and the SNACK signal (selected number acknowledge). See <a href="#">Figure 16.2.2 (b)</a> .
OPIN[32]	UI[17] PNSTROBE Active when the robot is in a remote condition (CMDENBL = ON)	The PNSTROBE input is the program number select strobe input signal. See <a href="#">Figure 16.2.2 (b)</a> .
OPIN[33]	UI[18] PROD_START Active when the robot is in a remote condition (CMDENBL = ON)	The PROD_START input, when used with PNS, will initiate execution of the selected program from the PNS lines. When used without PNS, PROD_START executes the selected program from the current cursor position. See <a href="#">Figure 16.2.2 (b)</a> .

Table 16.2.2 (d) User Operator Panel Output Signals

OPOUT[n]	UOP Output Signal	Description
OPOUT[16]	UO[1] CMDENBL	CMDENBL is the command enable output. This output indicates that the robot is in a remote condition. This signal goes on when the System Configuration screen setting Remote/Local setup: is REMOTE. This output only stays on when the robot is not in a fault condition. When SYSRDY is OFF, CMDENBL is OFF. See <a href="#">Figure 16.2.2 (a)</a> and <a href="#">Figure 16.2.2 (b)</a> . This signal goes on when the following conditions are <b>all satisfied</b> : <ul style="list-style-type: none"> <li>• Teach pendant disabled</li> <li>• System Configuration screen setting Remote/Local setup: is REMOTE</li> <li>• SFSPD input is ON</li> <li>• ENBL input is ON</li> <li>• \$RMT_MASTER system variable is 0</li> <li>• Not in single step mode</li> <li>• Mode selection switch is set to AUTO (when mode select switch is installed)</li> </ul>
OPOUT[17]	UO[2] SYSRDY	SYSRDY is the system ready output. This output indicates that the servo motors are turned ON.

OPOUT[n]	UOP Output Signal	Description
OPOUT[18]	UO[3] PROGRUN	PROGRUN is the program run output. This output turns on when a program is running. See <a href="#">Figure 16.2.2 (b)</a> .
OPOUT[19]	UO[4] PAUSED	PAUSED is the paused program output. This output turns on when a program is paused.
OPOUT[20]	UO[5] HELD	HELD is the hold output. This output turns on when the SOP HOLD button has been pressed, or the UOP *HOLD input is OFF.
OPOUT[21]	UO[6] FAULT	FAULT is the error output. This output turns on when a program is in an error condition.
OPOUT[22]	UO[7] ATPERCH	This output is the at perch output. This output turns on when the robot reaches the predefined perch position. When \$SHELL_WRK.\$KAREL_UOP=FALSE, then the system sets \$ATPERCH. The ATPERCH position = Reference position #1 .
OPOUT[23]	UO[8] TPENBL	TPENBL is the teach pendant enable output. This output turns on when the teach pendant is on.
OPOUT[24]	UO[9] BATALM	BATALM is the battery alarm output. This output turns on when the CMOS RAM battery voltage goes below 2.6 volts.
OPOUT[25]	UO[10] BUSY	BUSY is the processor busy output. This signal turns on when the robot is executing a program or when the processor is busy.
OPOUT[26] - OPOUT[29]	UO[11] - UO[14] ACK 1-4	ACK 1-4 are the acknowledge signals output 1 through 4. These signals turn on when the corresponding RSR signal is received. See <a href="#">Figure 16.2.2 (a)</a> .
OPOUT[26] OPOUT[33]	UO[11] - UO[18] SNO 1-8	SNO 1-8 are the signal number outputs. These signals carry the 8-bit representation of the corresponding PNS selected program number. If the program cannot be represented by an 8-bit number, the signal is set to all zeroes or off. See <a href="#">Figure 16.2.2 (b)</a> .
OPOUT[34]	UO[19] SNACK	SNACK is the signal number acknowledge output. This output is pulsed if the program is selected by PNS input. See <a href="#">Figure 16.2.2 (b)</a> .
OPOUT[35]	UO[20] RESERVED	-
OPOUT[36]	UO[21] UNCAL (option)	UNCAL is the uncalibrated output. This output turns on when the robot is not calibrated. The robot is uncalibrated when the controller loses the feedback signals from one or all of the motors. Set \$OPWORK.\$OPT_OUT = 1 to use this signal.
OPOUT[37]	UO[22] UPENBL (option)	UPENBL is the user panel enable output. This output indicates that the robot is in a remote condition. This signal goes on when the remote switch is turned to ON or when the ENBL input is received. <b>This output will stay on even if the robot is in a fault condition.</b> Set \$OPWORK.\$OPT_OUT = 1 to use this signal.
OPOUT[38]	UO[23] LOCKED (option)	-

OPOUT[n]	UOP Output Signal	Description
OPOUT[39]	UO[24] CSTOPO (option)	CSTOPO is the cycle stop output. This output turns on when the CSTOPI input has been received. Set \$OPWORK.\$OPT_OUT = 1 to use this signal.

### Timing Diagrams

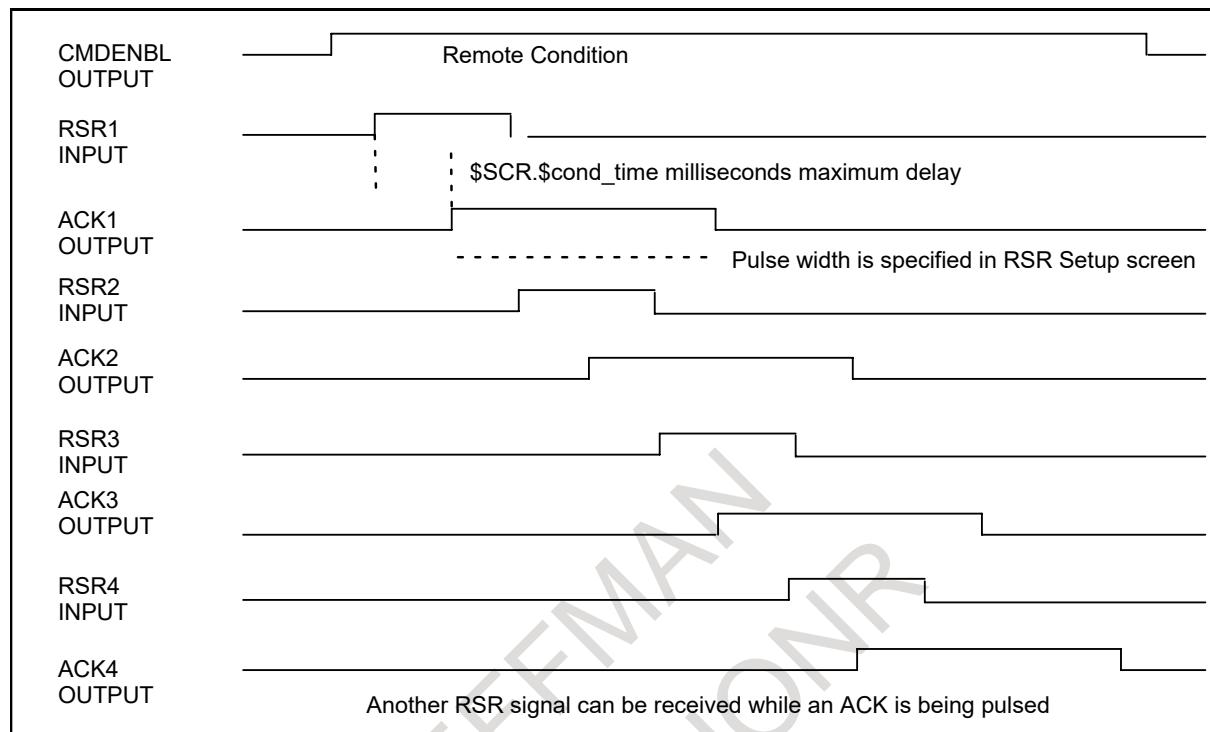


Figure 16.2.2 (a) RSR Timing Diagram

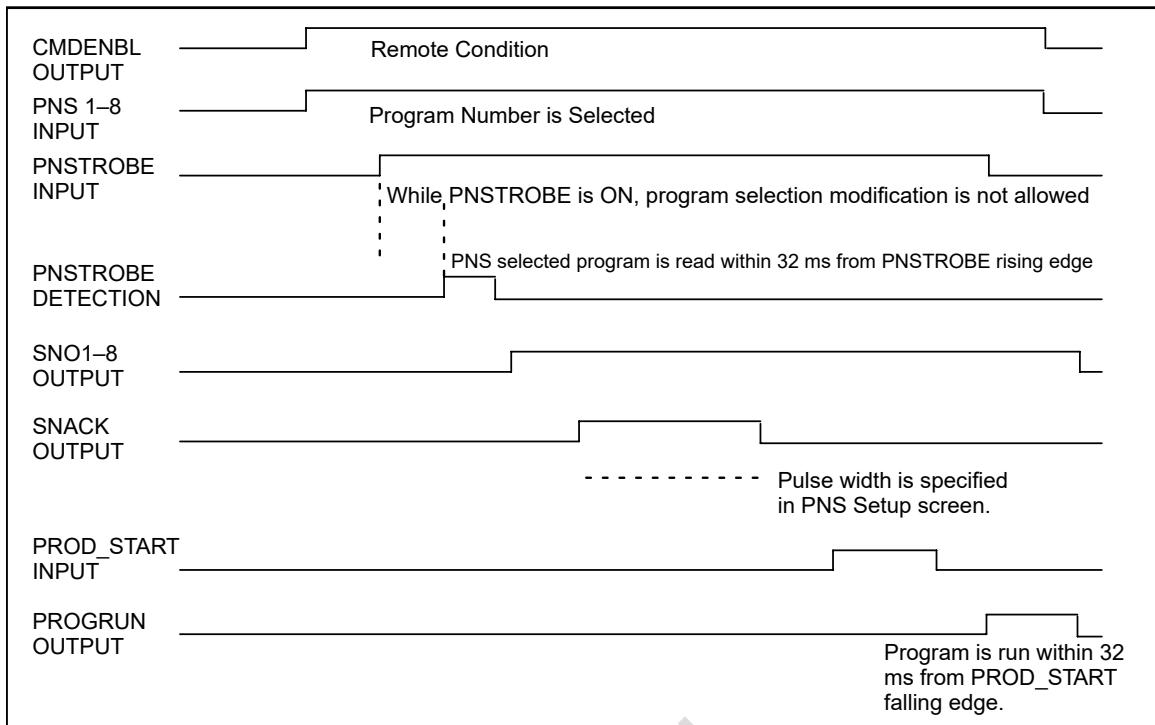


Figure 16.2.2 (b) PNS Timing Diagram

### 16.2.3 Teach Pendant Input and Output Signals (TPIN/TPOUT)

The teach pendant input signals (TPIN) provide read access to input signals generated by the teach pendant keys. Teach pendant inputs can be accessed through the TPIN port arrays. A KAREL program treats teach pendant input data as a BOOLEAN data type. The value is either ON (active – the key is pressed) or OFF (inactive – the key is not pressed). TPIN signals are accessed in KAREL programs by the name `TPIN[n]`, where `n` is the signal number, which is assigned internally. Refer to [Table 16.2.3](#) for teach pendant input signal assignments.

Table 16.2.3 Teach Pendant Input Signal Assignments

TPIN[n]	Teach Pendant Key
<b>EMERGENCY STOP AND DEADMAN</b>	
TPIN[250]	EMERGENCY STOP
TPIN[249]	ON/OFF switch
TPIN[247]	Right DEADMAN switch
TPIN[248]	Left DEADMAN switch
<b>Arrow Keys</b>	

TPIN[n]	Teach Pendant Key
TPIN[212]	Up arrow
TPIN[213]	Down arrow
TPIN[208]	Right arrow
TPIN[209]	Left arrow
TPIN[0]	Left and/or right shift
TPIN[204]	Shifted Up arrow
TPIN[205]	Shifted Down arrow
TPIN[206]	Shifted Right arrow
TPIN[207]	Shifted Left arrow
<b>Keypad Keys (shifted or unshifted)</b>	
TPIN[13]	ENTER
TPIN[8]	BACK SPACE
TPIN[48]	0
TPIN[49]	1
TPIN[50]	2
TPIN[51]	3
TPIN[52]	4
TPIN[53]	5
TPIN[54]	6
TPIN[55]	7
TPIN[56]	8
TPIN[57]	9
<b>Function Keys</b>	
TPIN[128]	PREV
TPIN[129]	F1
TPIN[131]	F2
TPIN[132]	F3
TPIN[133]	F4
TPIN[134]	F5
TPIN[135]	NEXT
TPIN[136]	Shifted PREV
TPIN[137]	Shifted F1
TPIN[138]	Shifted F2
TPIN[139]	Shifted F3
TPIN[140]	Shifted F4
TPIN[141]	Shifted F5
TPIN[142]	Shifted NEXT

TPIN[n]	Teach Pendant Key
<b>Menu Keys</b>	
TPIN[143]	SELECT
TPIN[144]	MENU
TPIN[145]	EDIT
TPIN[146]	DATA
TPIN[147]	FCTN
TPIN[148]	ITEM
TPIN[149]	+%
TPIN[150]	-%
TPIN[151]	HOLD
TPIN[152]	STEP
TPIN[153]	RESET
TPIN[240]	DISP
TPIN[203]	HELP
TPIN[154]	Shifted ITEM
TPIN[155]	Shifted +%
TPIN[156]	Shifted -%
TPIN[157]	Shifted STEP
TPIN[158]	Shifted HOLD
TPIN[159]	Shifted RESET
TPIN[227]	Shifted DISP
TPIN[239]	Shifted HELP
<b>User Function Keys</b>	
TPIN[173]	USER KEY 1
TPIN[174]	USER KEY 2
TPIN[175]	USER KEY 3
TPIN[176]	USER KEY 4
TPIN[177]	USER KEY 5
TPIN[178]	USER KEY 6
TPIN[210]	USER KEY 7
TPIN[179]	Shifted USER KEY 1
TPIN[180]	Shifted USER KEY 2
TPIN[181]	Shifted USER KEY 3
TPIN[182]	Shifted USER KEY 4
TPIN[183]	Shifted USER KEY 5
TPIN[184]	Shifted USER KEY 6
TPIN[211]	Shifted USER KEY 7

<b>TPIN[n]</b>	<b>Teach Pendant Key</b>
<b>Motion Keys</b>	

EFFMAN  
QUIRIONR

TPIN[n]	Teach Pendant Key
TPIN[185]	FWD
TPIN[186]	BWD
TPIN[187]	COORD
TPIN[28]	GROUP
TPIN[188]	+X
TPIN[189]	+Y
TPIN[190]	+Z
TPIN[191]	+X rotation
TPIN[192]	+Y rotation
TPIN[193]	+Z rotation
TPIN[12290]	+J7
TPIN[12294]	+J8
TPIN[194]	-X
TPIN[195]	-Y
TPIN[196]	-Z
TPIN[197]	-X rotation
TPIN[198]	-Y rotation
TPIN[199]	-Z rotation
TPIN[12292]	-J7
TPIN[12296]	-J8
TPIN[226]	Shifted FWD
TPIN[207]	Shifted BWD
TPIN[202]	Shifted COORD
TPIN[214]	Shifted +X
TPIN[215]	Shifted +Y
TPIN[216]	Shifted +Z
TPIN[217]	Shifted +X rotation
TPIN[218]	Shifted +Y rotation
TPIN[219]	Shifted +Z rotation
TPIN[12291]	Shifted +J7
TPIN[12295]	Shifted +J8
TPIN[220]	Shifted -X
TPIN[221]	Shifted -Y
TPIN[222]	Shifted -Z
TPIN[223]	Shifted -X rotation
TPIN[224]	Shifted -Y rotation
TPIN[225]	Shifted -Z rotation
TPIN[12293]	Shifted -J7
TPIN[12297]	Shifted -J8

TPIN[n]	Teach Pendant Key
<b>i Keys</b>	
TPIN[12288]	<i>i</i> key
TPIN[12298]	<i>i</i> MENU, Top Menu
TPIN[12299]	<i>i</i> SELECT, 4D Select Node Map
TPIN[12300]	<i>i</i> EDIT, 4D Position Register
TPIN[12301]	<i>i</i> DATA, 4D Edit Node Map
TPIN[12302]	<i>i</i> FCTN, Related Views
TPIN[12303]	<i>i</i> STEP
TPIN[12304]	<i>i</i> COORD
TPIN[12305]	<i>i</i> GROUP
TPIN[12306]	<i>i</i> +%
TPIN[12307]	<i>i</i> -%
TPIN[12308]	<i>i</i> Up arrow
TPIN[12309]	<i>i</i> Down arrow
TPIN[12310]	<i>i</i> Right arrow
TPIN[12311]	<i>i</i> Left arrow
TPIN[12312]	<i>i</i> DISP, Focus window
TPIN[12313]	<i>i</i> HELP
TPIN[12338]	<i>i</i> USER KEY 7, 4D Display
TPIN[12339]	<i>i</i> USER KEY 6
TPIN[12340]	<i>i</i> USER KEY 5
TPIN[12341]	<i>i</i> ENTER, space
TPIN[12314]	<i>i</i> +X
TPIN[12315]	<i>i</i> +Y
TPIN[12316]	<i>i</i> +Z
TPIN[12317]	<i>i</i> +X rotation
TPIN[12318]	<i>i</i> +Y rotation
TPIN[12319]	<i>i</i> +Z rotation
TPIN[12320]	<i>i</i> +J7 rotation
TPIN[12321]	<i>i</i> +J8 rotation
TPIN[12322]	<i>i</i> -X
TPIN[12323]	<i>i</i> -Y
TPIN[12324]	<i>i</i> -Z
TPIN[12325]	<i>i</i> -X rotation
TPIN[12326]	<i>i</i> -Y rotation
TPIN[12327]	<i>i</i> -Z rotation
TPIN[12328]	<i>i</i> -J7 rotation
TPIN[12329]	<i>i</i> -J8 rotation

Three teach pendant output signals are available for use:

- TPOUT[6] - controls teach pendant USER LED #1
- TPOUT[7] - controls teach pendant USER LED #2
- TPOUT[8] - controls teach pendant USER LED #3

## 16.3 SERIAL INPUT/OUTPUT

The serial I/O system allows you to communicate with peripheral serial devices connected to the KAREL system. For example, you could use serial I/O to write messages from one of the communications ports to a remote terminal across a cable that connects to the controller.

To use serial I/O you must provide a serial device and the appropriate cable. Refer to the Maintenance Manual, specific to your robot model, for electrical specifications.

The communications ports that you use to read and write serial data are defined in the system software. Each software port is associated with physical connectors on the controller to which you attach the communications cable.

Setting up a port means initializing controller serial ports to use specific devices, such as the CRT/KB. Initializing ports involves setting up specific information for a port based on the kind of device that will connect to the port. This is done on the teach pendant **PORT INIT** screen.

The controller supports up to four serial ports. Several different kinds of devices can be connected to these ports.

### NOTE

P3: and P4: are not available on the R-30iB Mate Plus Controller.

Figure 16.3 (a) and Figure 16.3 (b) show the ports on each controller.

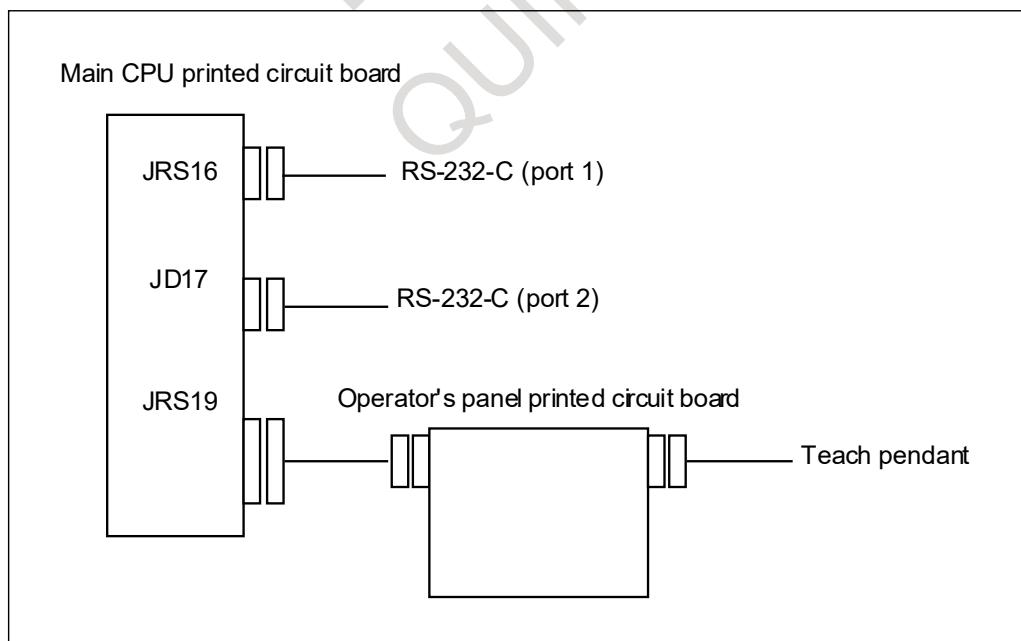
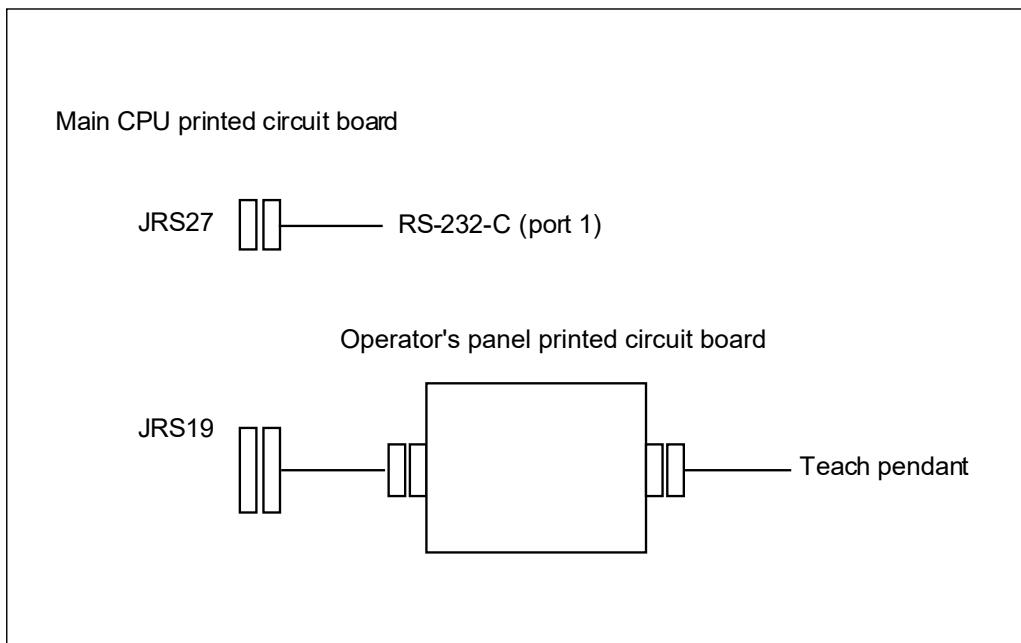


Figure 16.3 (a) R-30iB Plus Controller Ports

**Figure 16.3 (b) R-30iB Mate Plus Controller Ports****Ports**

Up to four ports are available, P1-P4. [Table 16.3 \(a\)](#) lists the ports. You can set up ports P2 through P4 if you have them, but you cannot set up the teach pendant port, P1.

**NOTE**

P3: and P4: are not available on the R-30iB Mate Plus Controller.

**Table 16.3 (a) Ports P1 - P4**

Port	Item Name on Screen	Kind of Port	Use	Default Device
P1	Teach Pendant  <b>NOTE</b>  This is a dedicated port and cannot be changed.	RS-422	Teach pendant	Teach pendant
P2	JRS16 RS-232-C	RS-232-C	Any device	Maintenance Console
P3	JD17 RS-232-C on Main CPU card	RS-232-C		KCL
P4	JD17 on Main CPU card. This port is displayed on the teach pendant if \$RS232_NPORT=4.	RS-422		No use

## Devices

You can modify the default communications settings for each port except port 1, which is dedicated to the teach pendant (TP). [Table 16.3 \(b\)](#) lists the default settings for each kind of device you can connect to a port.

**Table 16.3 (b) Default Communications Settings for Devices**

Device	Speed (baud)	Parity Bit	Stop Bit	Timeout Value (sec)
Sensor*	4800	Odd	1 bit	0
Host Comm.*	4800	Odd	1 bit	0
KCL/CRT	9600	None	1 bit	0
Maintenance Console	9600	None	1 bit	0
Factory Terminal	9600	None	1 bit	0
TP Demo Device	9600	None	1 bit	0
No Use	9600	None	1 bit	0
Current Position (for use with the Current Position option)	9600	None	1 bit	0
PMC Programmer	9600	None	2 bit	0
Modem/PPP	Refer to the <i>Internet Options Setup and Operations Manual (MAROUIN9010171E)</i> or the <i>Ethernet Function OPERATOR'S MANUAL (B-82974EN)</i> for information on the supported modems.			
HMI Device	19200	Odd	1 bit	0

\*You can adjust these settings; however, if you do, they might not function as intended because they are connected to an external device.

After the hardware has been connected and the appropriate port is configured and the external port is connected, you can use KAREL language OPEN FILE, READ, and WRITE statements to communicate with the peripheral device.

Higher levels of communication protocol are supported as an optional feature.

**See Also:** [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#) for more information on the statements and built-ins available in KAREL

Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)* for more information about setting up ports.

EFFMAN  
QUIRIONR

# 17 MULTI-TASKING

Multi-tasking allows more than one program to run on the controller on a time-sharing basis, so that multiple programs appear to run simultaneously.

Multi-tasking is especially useful when you are executing several sequences of operations which can generally operate independently of one another, even though there is some interaction between them. For example:

- A process of monitoring input signals and setting output signals.
- A process of generating and transmitting log information to a cell controller and receiving commands or other input data from a cell controller.

It is important to be aware that although multiple tasks seem to operate at the same time, they are sharing use of the same processor, so that at any instant only one task is really being executed. With the exception of interruptible statements, once execution of a statement is started, it must complete before statements from another task can be executed. The following statements are interruptible:

- READ
- DELAY
- WAIT
- WAIT FOR

Refer to [Section 17.4, TASK SCHEDULING](#) for information on how the system decides which task to execute first.

## 17.1 MULTI-TASKING TERMINOLOGY

The following terminology and expressions are used in this chapter.

### Task or User task

A task, or user task, is a user program that is running or paused. A task is executed by an *interpreter*. A task is created when the program is started and eliminated when the interpreter it is assigned to, becomes assigned to another task.

### Interpreter

An interpreter is a system component that executes user programs. At a cold or controlled start, ( $\$MAXNUMTASKS + 2$ ) interpreters are created. These interpreters are capable of concurrently executing tasks.

### Task name

Task name is the program name specified when the task is created. When you create a task, specify the name of the program to be executed as the task name.

#### NOTE

The task name does not change once the task is created.

Therefore, when an external routine is executing, the current executing program name is not the same as the task name. When you send any requests to the task, use the task name, not the current program name.

### Motion control

Motion control is defined by a bit mask that specifies the motion groups of which a task has control. Only one task at a time can control a motion group. However, different tasks can control different motion groups simultaneously. Refer to [Section 17.3, MOTION CONTROL](#), for more information.

## 17.2 INTERPRETER ASSIGNMENT

---

When a task is started, it is assigned to an interpreter. The interpreter it is assigned to (1, 2, 3, ...) determines its task number. The task number is used in PAUSE PROGRAM, ABORT PROGRAM and CONTINUE PROGRAM condition handler actions. The task number for a task can be determined using the GET\_TSK\_INFO built-in.

The following are rules for assigning a task to an interpreter:

- If the task is already assigned to an interpreter, it uses the same interpreter.
- A task is assigned to the first available interpreter that currently has no tasks assigned to it.
- If all interpreters are assigned to tasks, a new task will be assigned to the first interpreter that has an aborted task.
- If none of the above can be done, the task cannot be started.

## 17.3 MOTION CONTROL

---

An important restriction in multi-tasking is in the control of the various motion groups. Only one task can have control, or use of, a group of axes. A task requires control of the group(s) in the following situations:

- When the task starts, if the controller directive %NOLOCKGROUP is not used. If the %LOCKGROUP directive is not used, the task requires control of all groups by default. If %LOCKGROUP is used, control of the specified groups is required.

For teach pendant programs, motion control is required when the program starts, unless the **DETAIL** page from the **SELECT** screen is used to set the Group Mask to [\*, \*, \*, \*, \*].

- When a task executes the **LOCK\_GROUP** built-in, it requires the groups specified by the group mask.
- When a task calls a **ROUTINE** or teach pendant program, it requires control of those group(s). The group(s) required by a **ROUTINE** or TPP+ program are those specified, or implied, by controller directives or in the teach pendant **DETAIL** setup.

A task will be given control of the required group(s), assuming:

- No other task has control of the group.
- The teach pendant is not enabled, with the exception that motion control can be given to a program when it is started using **SHIFT-FWD** at the teach pendant or if it has the %TPMOTION directive.
- There are no emergency stops active.
- The servos are ready.
- The UOP signal IMSTP is not asserted.

A task will be paused if it is not able to get control of the required group(s).

After a task gets control of a group, it keeps it until one of the following:

- The task ends (aborts).
- The task executes the **UNLOCK\_GROUP** built-in.

- The task passes control of the group(s) in a RUN\_TASK built-in.
- The ROUTINE or teach pendant program returns, and groups were required by a ROUTINE or teach pendant program, but not by the calling program.

## 17.4 TASK SCHEDULING

---

A task that is currently running (not aborted or paused) will execute statements until one of the following:

- A hold condition occurs.
- A higher priority program becomes ready to run.
- The task time slice expires.
- The program aborts or pauses.

The following are examples of hold conditions:

- Waiting for a read operation to complete.
- Waiting for a motion to complete.
- Waiting for a WAIT, WAIT FOR, or DELAY statement to complete.

A task is ready to run when it is in running state and has no hold conditions. Only one task is actually executed at a time. There are two rules for determining which task will be executed when more than one task is ready to run:

- Priority - If two or more tasks of different priority are ready to run, the task with higher priority is executed first. Refer to [Section 17.4.1, Priority Scheduling](#) for more information.
- Time-slicing - If two tasks of the same priority are ready to run, execution of the tasks is time-sliced. Refer to [Section 17.4.2, Time Slicing](#) for more information.

### 17.4.1 Priority Scheduling

---

If two or more tasks with different priorities are ready to run, the task with the highest priority will run first. The priority of a task is determined by its priority number. Priority numbers must be in the range from -8 to 143. The lower the priority number, the higher the task priority.

For example: if TASK\_A has a priority number of 50 and TASK\_B has a priority number of 60, and both are ready to run, TASK\_A will execute first, as long as it is ready to run.

A task priority can be set in one of the following ways:

- By default, each user task is assigned a priority of 50.
- KAREL programs may contain the %PRIORITY translator directive.
- The SET\_TSK\_ATTR built-in can be used to set the current priority of any task.

In addition to affecting other user tasks, task priority also affects the priority of the interpreter executing it, relative to that of other system functions. If the user task has a higher priority (lower priority number) than the system function, as long as the user task is ready to run, the system function will not be executed. The range of user task priorities is restricted at the high priority end. This is done so that the user program cannot interfere with motion interpolation. Motion interpolation refers to the updates required to cause a motion to complete.

The following table indicates the priority of some other system functions.

**Table 17.4.1 System Function Priority Table**

<b>Priority</b>	<b>System Function</b>	<b>Effect of Delaying Function</b>
-8	Maximum priority	New motions delayed.
-1	Motion Planner	New motions delayed.
4	TP Jog	Jogging from the Teach Pendant delayed.
54	Error Logger	Update of system error log delayed.
73	KCL	Execution of KCL commands delayed.
82	CRT manager	Processing of CRT soft-keys delayed.
88	TP manager	General teach pendant activity delayed.
143	Lowest priority	Does not delay any of the above.

## 17.4.2 Time Slicing

---

If two or more tasks of the same priority are ready to run, they will share the system resources by time-slicing, or alternating use of the system.

A time-slice permits other tasks of the same priority to execute, but not lower priority tasks.

The default time-slice for a task is 256 msec. Other values can be set using the %TIMESLICE directive or the SET\_TASK\_ATTR built-in.

## 17.5 STARTING TASKS

---

There are a number ways to start a task.

- KCL RUN command. Refer to the [Section C.52, RUN command](#) in [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#).
- Operator Panel start key. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).
- User operator panel start signal. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN)
- User operator panel PNS signal. Refer to [Section 17.5.1, Running Programs from the User Operator Panel \(UOP\) PNS Signal](#) for more information.
- Teach pendant SHIFT-FWD key. Refer to your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN) for more information.
- Teach pendant program executes a RUN instruction. Refer to [Section 17.5.2, Child Tasks](#) for more information.
- KAREL program executes the RUN\_TASK built-in. Refer to [Section 17.5.2, Child Tasks](#) for more information.

In each case, the task will not start running if it requires motion control that is not available.

## 17.5.1 Running Programs from the User Operator Panel (UOP) PNS Signal

---

A program is executed:

- If the binary value of the UOP PNS signals is non-zero and the UOP PROGSTART signal is asserted
- If there is currently a program with the name “PNSnnnn,” where nnnn is the decimal value of the PNS signals plus the current value of `$$HELLCFG.$jobbase`.

A program is not executed:

- If the binary value of the PNS signals is zero.

Multiple programs can be started in this way, as long as there is no motion group overlap.

If the task name determined from the PNS is in a paused state, the PROGSTART signal is interpreted as a CONTINUE signal. If `$$HELLCFG.$contonly` is TRUE, this is the only function of the PNS/PROGSTART signals.

If `$$HELLCFG.$useabort` is TRUE, the PNS signals can be used to abort a running task. The name of the task to be aborted is the same as that used with the PROGSTART signal. In this case, abort is triggered by the UOP CSTOPI signal

## 17.5.2 Child Tasks

---

A running task can create new tasks. This new task is called a child task. The task requesting creation of the child task is called the parent task. In teach pendant programs, a new task is created by executing a RUN instruction. In KAREL programs a new task can be created using the RUN\_TASK built-in. The parent and child task may not require the same motion group.

Once a child task is created, it runs independently of its parent task, with the following exception:

- If a parent task is continued and its child task is paused, the child task is also continued.
- If a parent task is put in STEP mode, the child task is also put in STEP mode.

If you want the child task to be completely independent of the parent, a KAREL program can initiate another task using the KCL or KCL\_NOWAIT built-ins to issue a KCL>RUN command.

# 17.6 TASK CONTROL AND MONITORING

---

There are three environments from which you can control and monitor tasks:

1. Teach Pendant Programs (TPP) - [Section 17.6.1, From TPP Programs](#)
2. KAREL Programs - [Section 17.6.2, From KAREL Programs](#)
3. KCL commands - [Section 17.6.3, From KCL](#)

## 17.6.1 From TPP Programs

---

The RUN program instruction can be used to run another task. The RESUME\_PROG program instruction can be used to continue a paused task. Refer to your application-specific *Setup and Operations Manual* or

the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)* for information about the RUN and RESUME\_PROG instructions.

## 17.6.2 From KAREL Programs

---

There are a number of built-ins used to control and monitor other tasks. See the description of these built-ins in [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#).

- RUN\_TASK executes a task.
- CONT\_TASK resumes execution of a PAUSED task.
- PAUSE\_TASK pauses a task.
- ABORT\_TASK aborts a task.
- CONTINUE condition handler action causes execution of a task.
- ABORT condition handler action causes a task to be aborted.
- PAUSE condition handler action causes a task to be paused.
- GET\_TSK\_INFO determines whether a specified task is running, paused, or aborted. Also determines what program and line number is being executed, and what, if anything, the task is waiting for.

## 17.6.3 From KCL

---

The following KCL commands can be used to control and monitor the status of tasks. Refer to [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#) for more information.

- RUN <task\_name> starts or continues a task.
- CONT <task\_name> continues a task.
- PAUSE <task\_name> pauses a task.
- ABORT <task\_name> aborts a task.
- SHOW TASK <task\_name> displays the status of a task.
- SHOW TASKS displays the status of all tasks.

## 17.7 USING SEMAPHORES AND TASK SYNCHRONIZATION

---

Good design dictates that separate tasks be able to operate somewhat independently. However, they should also be able to interact.

The KAREL controller supports counting semaphores. The following operations are permitted on semaphores:

### Clear a semaphore

(KAREL: CLEAR\_SEMA built-in): sets the semaphore count to zero.

All semaphores are cleared at cold start. It is good practice to clear a semaphore prior to using it. Before several tasks begin sharing a semaphore, one and only one of these tasks, should clear the semaphore.

### Post to a semaphore

(KAREL: POST\_SEMA built-in): adds one to the semaphore count.

If the semaphore count is zero or greater, when the post semaphore is issued, the semaphore count will be incremented by one. The next task waiting on the semaphore will decrement the semaphore count and continue execution. Refer to [Figure 17.7 \(a\)](#).

If the semaphore count is negative, when the post semaphore is issued, the semaphore count will be incremented by one. The task which has been waiting on the semaphore the longest will then continue execution. Refer to [Figure 17.7 \(a\)](#).

### Read a semaphore

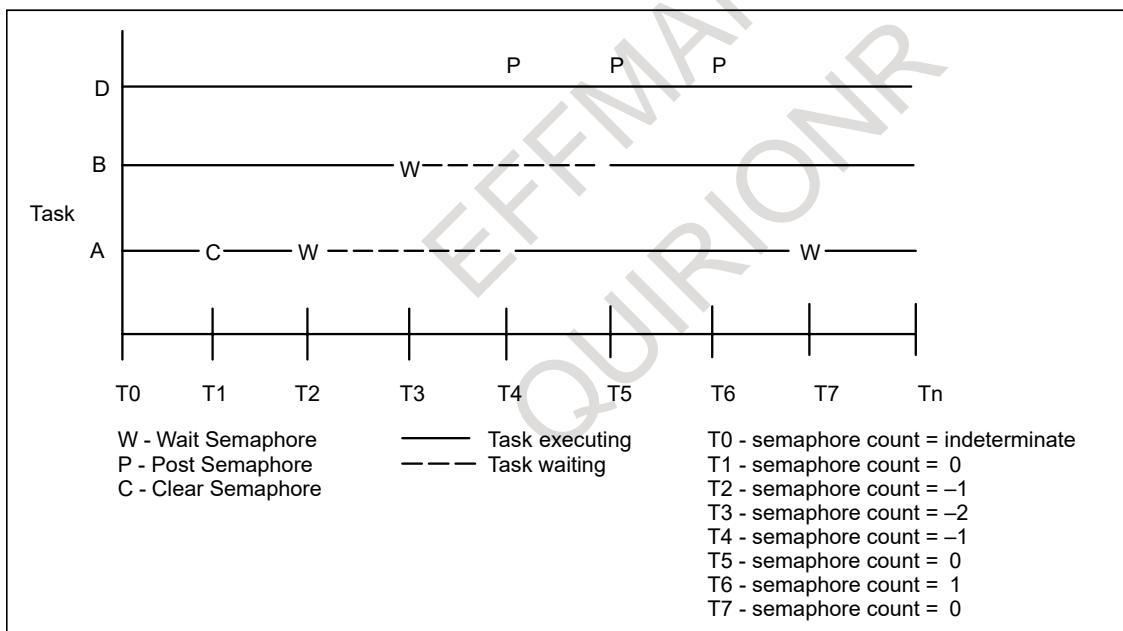
(KAREL: SEMA\_COUNT built-in): returns the current semaphore count.

### Wait for a semaphore

(KAREL: PEND\_SEMA built-in, SIGNAL SEMAPHORE action):

If the semaphore count is greater than zero when the wait semaphore is issued, the semaphore count will be decremented and the task will continue execution. Refer to [Figure 17.7 \(a\)](#).

If the semaphore count is less than or equal to zero (negative), the wait semaphore will decrement the semaphore count and the task will wait to be released by a post semaphore. Tasks are released on a first-in/first-out basis. For example, if *task A* waits on semaphore 1, then *task B* waits on semaphore 1. When *task D* posts semaphore 1, only *task A* will be released. Refer to [Figure 17.7 \(a\)](#).



**Figure 17.7 (a) Task Synchronization Using a Semaphore**

### Example:

Semaphores can be used to implement a task that acts as a request server. In the following example, the main task waits for the server to complete its operation. Semaphore[4] is used to control access to rqst\_param or R[5]. Semaphore[5] is used to signal the server task that service is being requested; semaphore[6] is used by the server to signal that the operation is complete.

The main task would contain the following KAREL:

**Figure 17.7 (b) Main Task**

```
--KAREL
CLEAR_SEMA(4)
CLEAR_SEMA(5)
CLEAR_SEMA(6)
RUN_TASK(`server',0,TRUE,TRUE,1,STATUS)
PEND_SEMA(4,max_time,time_out)
rqst_param=10
POST_SEMA(5)
PEND_SEMA(6,max_time,time_out)
```

The server task would contain the following KAREL code:

**Figure 17.7 (c) Server Task**

```
--KAREL
POST_SEMA(4)
WHILE TRUE DO
    PEND_SEMA(5,max_time,time_out)
    IF rqst_param=10 THEN
        do_something
    ENDIF
    POST_SEMA(4)
    POST_SEMA(6)
ENDWHILE
```

### Example

The program example in [Figure 17.7 \(d\)](#) thru [Figure 17.7 \(f\)](#) shows how semaphores and tasks can be used together for synchronization. MAIN\_TASK.KL is used to initialize the semaphore (MOTION\_CTRL) and then runs both TASK\_A.KL and TASK\_B.KL. MAIN\_TASK.KL then waits for TASK\_A and TASK\_B to abort before completing. TASK\_A waits until you press F1 and then moves the robot to the HOME position. TASK\_B waits until you press F2 and then moves the robot along a path.

**Figure 17.7 (d) Semaphore and Task Synchronization Program Example - MAIN TASK**

```
PROGRAM main_task
%nolockgroup
VAR
    motion_ctrl: INTEGER
    tsk_a_done : BOOLEAN
    tsk_b_done : BOOLEAN
    tmr        : INTEGER
    status      : INTEGER
-----
--  INIT_LOCK: Initialize the semaphore      --
--            to make sure its count is at   --
--            zero before using it. Then    --
--            post this semaphore which will --
--            allow the first pend to the   --
--            semaphore to continue       --
--            execution.                  --
-----
ROUTINE init_lock
BEGIN
```

```

CLEAR_SEMA (motion_ctrl) -- makes sure semaphore is zero before
-- using it.
POST_SEMA (motion_ctrl) -- makes motion_ctrl available immediately
END init_lock
-----
-- IS_TSK_DONE : Find out if the specified --
-- task is running or not. --
-- If the task is aborted then --
-- return TRUE otherwise FALSE.--
-----
ROUTINE is_tsk_done (task_name:STRING): BOOLEAN
VAR
    status : INTEGER      -- The status of the operation of GET_TSK_INFO
    task_no : INTEGER     -- Receives the current task number for
                          -- task name
    attr_out: INTEGER     -- Receives the TSK_STATUS output
    dummy   : STRING[2]   -- Does not receive any information
BEGIN
    GET_TSK_INFO (task_name, task_no, TSK_STATUS, attr_out, dummy,
    status)
    IF (attr_out = PG_ABORTED) THEN
        RETURN (TRUE) -- If task is aborted then return TRUE
    ENDIF
    RETURN(FALSE) -- otherwise task is not aborted and return
    FALSE
END is_tsk_done
BEGIN
    motion_ctrl = 1 -- Semaphore to allow motion control
    init_lock -- Make sure this is done just once
    FORCE_SPMENU ( tp_panel, spi_tpuser, 1) -- Force the Teach Pendant
                                              -- user screen to be seen
    RUN_TASK('task_a', 1, FALSE, FALSE, 1, status) -- Run task_a
    RUN_TASK('task_b', 1, FALSE, FALSE, 1, status) -- Run task_b
    REPEAT
        tsk_a_done = is_tsk_done ('task_a')
        tsk_b_done = is_tsk_done ('task_b')
        delay (100)
    UNTIL (tsk_a_done and tsk_b_done) -- Repeat until both task_a
END main_task -- and task_b are aborted

```

**Figure 17.7 (e) Semaphore and Task Synchronization Program Example - TASK A**

```

PROGRAM task_a
%nolockgroup
VAR
    motion_ctrl FROM main_task: INTEGER
    home_pos           : XYZWPR
    status             : INTEGER
-----
-- RUN_HOME : Lock the robot motion --
-- control. This task is --
-- moving the robot and must --
-- have control. --
-----
ROUTINE run_home
VAR
    time_out: BOOLEAN

```

```

BEGIN
    PEND_SEMA(motion_ctrl,-1,time_out)-- lock motion_ctrl from other
                                         -- tasks
                                         -- keep other tasks from moving robot
    LOCK_GROUP (1, status)
    SET_POS_REG(1, home_pos, status)
    do_move -- call TP program to move to the home position
    UNLOCK_GROUP (1, status)
    POST_SEMA(motion_ctrl) -- unlock motion_ctrl
                                         -- allow other task to move robot
END run_home
BEGIN
    set_cursor (tpfunc, 1, 4, status)
    write tpfunc ('HOME',CR)
    wait for TPIN[129]+ -- wait for F1 to be pressed
    run_home
END task_a

```

**Figure 17.7 (f) Semaphore and Task Synchronization Program Example - TASK B**

```

PROGRAM task_b
%nolockgroup
VAR
    motion_ctrl FROM main_task : INTEGER
    work_path : PATH
    move_pos : XYZWPR
    status : INTEGER
-----
-- do_work      : Lock the robot from other --
--                 tasks and do work. This --
--                 task is doing motion and --
--                 must lock motion control so --
--                 that another task does not --
--                 try to do motion at the --
--                 same time.
-----
ROUTINE do_work
VAR
    time_out: BOOLEAN
BEGIN
    PEND_SEMA (motion_ctrl,-1,time_out) -- lock motion_ctrl from other
                                         -- tasks keep other tasks from
                                         -- moving robot
    LOCK_GROUP (1, status)
    FOR i=1 to PATH_LEN(work_path) DO
        move_pos = work_path[i]
        SET_POS_REG(1, move_pos, status)
        do_move -- call TP program to move to path node
    ENDFOR
    UNLOCK_GROUP (1, status)
    POST_SEMA(motion_ctrl) -- unlock motion_ctrl allow
                                         -- other task to move robot
END do_work
BEGIN
    set_cursor(tpfunc, 1, 10, status)
    write tpfunc('WORK',CR)
    wait for TPIN[131]+ -- wait until F2 is pressed

```

```

do_work
END task_b

```

## 17.8 USING QUEUES FOR TASK COMMUNICATIONS

Queues are supported only in KAREL. A queue is a first-in/first-out list of integers. They are used to pass information to another task sequentially. A queue consists of a user variable of type QUEUE\_TYPE and an ARRAY OF INTEGER. The maximum number of entries in the queue is determined by the size of the array.

The following operations are supported on queues:

- INIT\_QUEUE: initializes a queue and sets it to empty.
- APPEND\_QUEUE: adds an integer to the list of entries in the queue.
- GET\_QUEUE: reads the oldest (top) entry from the queue and deletes it.

These, and other built-ins related to queues (DELETE\_QUEUE, INSERT\_QUEUE, COPY\_QUEUE) are described in [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#).

A QUEUE\_TYPE data type has one user accessible element, n\_entries. This is the number of entries that have been added to the queue and not read out. The array of integer used with a queue, is used by the queue built-ins and should not be referenced by the KAREL program.

### Example

The following example illustrates a more powerful request server, in which more than one task is posting requests and the requester does not wait for completion of the request.

The requester would contain the following code:

**Figure 17.8 (a) Requester**

```

--declarations
VAR
    rqst_queue FROM server: QUEUE_TYPE
    rqst_data FROM server: ARRAY[100] OF INTEGER
    status: INTEGER
    seq_no: INTEGER
    -- posting to the queue --
    APPEND_QUEUE (req_code, rqst_queue, rqst_data, seq_no, status)

```

The server task would contain the following code:

**Figure 17.8 (b) Server**

```

PROGRAM server
VAR
    rqst_queue: QUEUE_TYPE
    rqst_data : ARRAY[100] OF INTEGER
    status      : INTEGER
    seq_no     : INTEGER
    rqst_code : INTEGER
BEGIN
    INIT_QUEUE(rqst_queue)    --initialization
    WHILE TRUE DO             --serving loop
        WAIT FOR rqst_code.n_entries > 0
        GET_QUEUE (rqst_queue, rqst_data, rqst_code, seq_no, status)

```

```
SELECT rqst_code OF
CASE (1) : do_something
ENDSELECT
ENDWHILE
END server
```

EFFMAN  
QUIRIONR

# **APPENDIX**

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

# A KAREL LANGUAGE ALPHABETICAL DESCRIPTION

---



---

This appendix describes, in alphabetical order, each standard KAREL language element, including:

- Data types
- Executable statements and clauses
- Condition handler conditions and actions
- Built-in routines
- Translator directives

A brief example of a typical use of each element is included in each description.

## NOTE

If, during program execution, any uninitialized variables are encountered as arguments for built-in routines, the program pauses and an error is displayed. Either initialize the variable, or abort the program.

## Conventions

This section describes each standard element of the KAREL language in alphabetical order. Each description includes the following information:

**Purpose:** Indicates the specific purpose the element serves in the language

**Syntax:** Describes the proper syntax needed to access the element in KAREL. [Table A \(a\)](#) describes the syntax notation that is used.

**Table A (a) Syntax Notation**

Syntax	Meaning	Example	Result
< >	Enclosed words are optional	AAA <BBB>	AAA AAA BBB
{ }	Enclosed words are optional and can be repeated	AAA {BBB}	AAA AAA BBB AAA BBB BBB AAA BBB BBB BBB
	Separates alternatives	AAA   BBB	AAA BBB
<   >	Separates an alternative if only one or none can be used	AAA <BBB   CCC>	AAA AAA BBB AAA CCC
	Exactly one alternative must be used	AAA    BBB   CCC	AAA BBB AAA CCC

Syntax	Meaning	Example	Result
{   }	Any combination of alternatives can be used	AAA { BBB   CCC }	AAA AAA BBB AAA CCC AAA BBB CCC AAA CCC BBB AAA BBB CCC BBB BBB
< <   > >	Nesting of symbols is allowed. Look at the innermost notation first to see what it describes, then look at the next innermost layer to see what it describes, and so forth.	AAA < BBB < CCC   DDD >	AAA AAA BBB AAA BBB CCC AAA BBB DDD

If the built-in is a function, the following notation is used to identify the data type of the value returned by the function:

Function Return Type: data\_type

Input and output parameter data types for functions and procedures are identified as:

[in] param\_name: data\_type

[out] param\_name: data\_type

where :

[in] specifies the data type of parameters which are passed into the routine

[out] specifies the data type of parameters which are passed back into the program from the routine

%ENVIRONMENT Group specifies the %ENVIRONMENT group for built-in functions and procedures, which is used by the off-line translator. Valid values are: BYNAM, CTDEF, ERRS, FDEV, FLBT, IOSETUP, KCL, MEMO, MIR, MOTN, MULTI, PATHOP, PBQMGR, REGOPE, STRNG, SYSDEF, TIM, TPE, TRANS, UIF, VECTR. The SYSTEM group is automatically used by the off-line translator.

**Details:** Lists specific rules that apply to the language element. An italics-type font is used to denote keywords input by the user within the syntax of the element.

**See Also:** Refers the reader to places in the document where more information can be found.

**Example:** Displays a brief example and explanation of the element.

## KAREL Language Elements Summary

Figure A (a) through Figure A (f) list the KAREL language elements, described in this appendix, by the type of element. Figure A (g) lists these elements in alphabetical order.

ABORT Action  
Assignment Action  
CANCEL Action  
CONTINUE Action  
DISABLE CONDITION Action  
ENABLE CONDITION Action

```
HOLD Action
NOABORT Action
NOMESSAGE Action
NOPAUSE Action
PAUSE Action
Port_Id Action
PULSE Action
RESUME Action
SIGNAL EVENT Action
SIGNAL SEMAPHORE Action
STOP Action
UNHOLD Action
UNPAUSE Action
```

**Figure A (a) Actions**

```
EVAL Clause
FROM Clause
IN Clause
WHEN Clause
WITH Clause
```

**Figure A (b) Clauses**

```
ABORT Condition
CONTINUE Condition
ERROR Condition
EVENT Condition
PAUSE Condition
Port_Id Condition
Relational Condition
SEMAPHORE Condition
```

**Figure A (c) Conditions**

```
ARRAY Data Type
BOOLEAN Data Type
BYTE Data Type
CONFIG Data Type
DISP_DAT_T Data Type
FILE Data Type
INTEGER Data Type
JOINTPOS Data Type
PATH Data Type
POSITION Data Type
QUEUE_TYPE Data Type
REAL Data Type
SHORT Data Type
STD_PTH_NODE Data Type
STRING Data Type
STRUCTURE Data Type
VECTOR Data Type
XYZWPR Data Type
XYZWPREXT Data Type
```

**Figure A (d) Data Types**

```
%ALPHABETIZE
%CMOSVARS
```

```
%CMOS2SHADOW
%COMMENT
%CRTDEVICE
%DEFGROUP
%DELAY
%ENVIRONMENT
%INCLUDE
%LOCKGROUP
%NOABORT
%NOBUSYLAMP
%NOLOCKGROUP
%NOPAUSE
%NOPAUSESHFT
%PRIORITY
%SHADOWVARS
%STACKSIZE
%TIMESLICE
%TPMOTION
%UNINITVARS
```

**Figure A (e) Directives**

**Table A (b) KAREL Built-In Routine Summary**

Category	Identifier		
<b>Byname</b>	CALL_PROG CALL_PROGLIN	CURR_PROG FILE_LIST	PROG_LIST VAR_INFO VAR_LIST
<b>Data Acquisition</b>	DAQ_CHECKP DAQ_REGPIPE	DAQ_START DAQ_STOP	DAQ_UNREG DAQ_WRITE
<b>Data Transfer Between Robots over Ethernet</b>	RGET_PORTCMT RGET_PORTSIM RGET_PORTVAL RGET_PREGCMT RGET_REG RGET_REG_CMT RGET_SREGCMT	RGET_STR_REG RNREG_RECV RNREG_SEND RPREG_RECV RPREG_SEND RSET_INT_REG RSET_PORTCMT	RSET_PORTSIM RSET_PORTVAL RSET_PREGCMT RSET_REALREG RSET_REG_CMT RSET_SREGCMT RSET_STR_REG
<b>Error Code Handling</b>	ERR_DATA	POST_ERR	POST_ERR_L
<b>File and Device Operation</b>	CHECK_NAME COMPARE_FILE COPY_FILE DELETE_FILE DISMOUNT_DEV DOSFILE_INF	FORMAT_DEV MOUNT_DEV MOVE_FILE PRINT_FILE PURGE_DEV RENAME_FILE	XML_ADDTAG XML_GETDATA XML_REMTAG XML_SCAN XML_SETVAR
<b>iPhone Communications</b>	RMCN_ALERT	RMCN_SEND	
<b>KCL Operation</b>	KCL	KCL_NO_WAIT	KCL_STATUS

Category	Identifier		
<b>Memory Operation</b>	CLEAR CREATE_VAR LOAD LOAD_STATUS	PROG_BACKUP PROG_CLEAR PROG_RESTORE RENAME_VAR	RENAME_VARS SAVE SAVE_DRAM
<b>Mirror</b>	MIRROR		
<b>Motion and Program Control</b>	CNCL_STP_MTN	MOTION_CTL	RESET
<b>Multi-programming</b>	ABORT_TASK CLEAR_SEMA CONT_TASK GET_TSK_INFO LOCK_GROUP	PAUSE_TASK PEND_SEMA POST_SEMA RUN_TASK	SEMA_COUNT SET_TSK_ATTR SET_TSK_NAME UNLOCK_GROUP
<b>Path Operation</b>	APPEND_NODE COPY_PATH	DELETE_NODE INSERT_NODE	NODE_SIZE PATH_LEN
<b>Personal Computer Communications</b>	ADD_BYNAMEPC ADD_INTPC	ADD_REALPC ADD_STRINGPC	SEND_DATAPC SEND_EVENTPC
<b>Position</b>	CHECK_EPOS CNV_JPOS_REL CNV_REL_JPOS CURPOS CURJPOS	FRAME IN_RANGE J_IN_RANGE JOINT2POS	POS POS2JOINT SET_PERCH UNPOS
<b>Process I/O Setup</b>	CLR_PORT_SIM GET_PORT_ASG GET_PORT_CMT GET_PORT_MOD	GET_PORT_SIM GET_PORT_VAL IO_MOD_TYPE SET_PORT_ASG	SET_PORT_CMT SET_PORT_MOD SET_PORT_SIM SET_PORT_VAL
<b>Queue Manager</b>	APPEND_QUEUE COPY_QUEUE DELETE_QUEUE	GET_QUEUE INIT_QUEUE	INSERT_QUEUE MODIFY_QUEUE
<b>Register Operation</b>	CLR_POS_REG GET_JPOS_REG GET_POS_REG GET_PREG_CMT GET_REG GET_REG_CMT	GET_SREG_CMT GET_STR_REG POS_REG_TYPE SET_EPOS_REG SET_INT_REG SET_JPOS_REG	SET_POS_REG SET_PREG_CMT SET_REAL_REG SET_REG_CMT SET_SREG_CMT SET_STR_REG

Category	Identifier		
<b>Serial I/O, File Usage</b>	BYTES_AHEAD BYTES_LEFT CLR_IO_STAT GET_FILE_POS GET_PORT_ATR	IO_STATUS MSG_CONNECT MSG_DISCO MSG_PING PIPE_CONFIG	SET_FILE_ATR SET_FILE_POS SET_PORT_ATR VOL_SPACE
<b>String Operation</b>	CNV_CNF_STRG CNV_CONF_STR CNV_INT_STR	CNV_REAL_STR CNV_STR_CONF CNV_STR_INT	CNV_STR_REAL STR_LEN SUB_STR
<b>System</b>	ABS ACOS ARRAY_LEN ASIN ATAN2 BYNAME CHR	COS EXP GET_VAR INDEX INV LN ORD	ROUND SET_VAR SIN SQRT TAN TRUNC UNINIT
<b>Time-of-Day Operation</b>	CNV_STR_TIME CNV_TIME_STR	GET_TIME GET_USEC_SUB	GET_USEC_TIM SET_TIME
<b>TPE Program</b>	AVL_POS_NUM CLOSE_TPE COPY_TPE CREATE_TPE DEL_INST_TPE GET_ATTR_PRG GET_JPOS_TPE	GET_POS_FRM GET_POS_TPE GET_POS_TYP GET_TPE_CMT GET_TPE_PRM OPEN_TPE SELECT_TPE	SET_ATTR_PRG SET_EPOS_TPE SET_JPOS_TPE SET_POS_TPE SET_TPE_CMT SET_TRNS_TPE
<b>Translate</b>	TRANSLATE		

Category	Identifier		
<b>User Interface</b>	ACT_SCREEN	DET_WINDOW	INI_DYN_DISS
	ADD_DICT	DISCTRL_ALPH	INIT_TBL
	ACT_TBL	DISCTRL_FORM	POP_KEY_RD
	ATT_WINDOW_D	DISCTRL_LIST	PUSH_KEY_RD
	ATT_WINDOW_S	DISCTRL_PLMN	READ_DICT
	CHECK_DICT	DISCTRL_SBMN	READ_DICT_V
	CNC_DYN_DISB	DISCTRL_TBL	READ_KB
	CNC_DYN_DISE	FORCE_LINK	REMOVE_DICT
	CNC_DYN_DISI	FORCE_SPMENU	SET_CURSOR
	CNC_DYN_DISP	INI_DYN_DISB	SET_LANG
	CNC_DYN_DISR	INI_DYN_DISE	WRITE_DICT
	CNC_DYN_DISS	INI_DYN_DISI	WRITE_DICT_V
	DEF_SCREEN	INI_DYN_DISP	
	DEF_WINDOW	INI_DYN_DISR	
<b>Vector</b>	APPROACH	ORIENT	
<b>Vision Operation</b>	V_ACQ_VAMAP	V_GET_OFFSET	VT_CLR_QUEUE
	V_ADJ_2D	V_GET_PASSFL	VT_DELETE_PQ
	V_CAM_CALIB	V_GET_READ	VT_GET_AREID
	V_CAM_CHECK	V_GET_VPARAM	VT_GET_FOUND
	V_CLR_VAMAP	V_IRCONNECT	VT_GET_LINID
	V_CSAPI_GETVALUE	V_LED_OFF	VT_GET_PFRT
	V_CSAPI_NUMSET	V_LED_ON	VT_GET_QUEUE
	V_CSAPI_RESETDATA	V_OVERRIDE	VT_GET_TIME
	V_CSAPI_SAVEDATA	V_RUN_FIND	VT_GET_TRYID
	V_CSAPI_SETVALUE	V_SAVE_IMREG	VT_PUT_QUE2
	V_CSAPI_TESTRUN	V_SET_REF	VT_READ_PQ
	V_DISPLAY4D	V_SNAP_VIEW	VT_SET_FLAG
	V_FIND_VIEW	VREG_FND_POS	VT_SET_LDBAL
	V_FIND_VLINE	VREG_OFFSET	VT_WRITE_PQ
	V_GET_FOUND	VT_ACK_QUEUE	

CR Input/Output Item

Figure A (f) Items

ABORT Statement  
Assignment Statement  
ATTACH Statement  
CANCEL Statement

```
CANCEL FILE Statement
CLOSE FILE Statement
CLOSE HAND Statement
CONDITION..ENDCONDITION Statement
CONNECT TIMER Statement
DELAY Statement
DISABLE CONDITION Statement
DISCONNECT TIMER Statement
ENABLE CONDITION Statement
FOR..ENDFOR Statement
GO TO Statement
HOLD Statement
IF..ENDIF Statement
OPEN FILE Statement
OPEN HAND Statement
PAUSE Statement
PROGRAM Statement
PULSE Statement
PURGE CONDITION Statement
READ Statement
RELAX HAND Statement
RELEASE Statement
REPEAT...UNTIL Statement
RESUME Statement
RETURN Statement
ROUTINE Statement
SELECT..ENDSELECT Statement
SIGNAL EVENT Statement
STOP Statement
UNHOLD Statement
USING..ENDUSING Statement
WAIT FOR Statement
WHILE..ENDWHILE Statement
WRITE Statement
```

Figure A (g) Statements

## A.1 - A - KAREL LANGUAGE DESCRIPTION

### A.1.1 ABORT Action

**Purpose:** Aborts execution of a running or paused task

**Syntax:** ABORT <PROGRAM[n]>

**Details:**

- If task execution is running or paused, the ABORT action will abort task execution.
- The ABORT action can be followed by the clause PROGRAM[n], where n is the task number to be aborted. Use GET\_TASK\_INFO to get a task number.
- If PROGRAM[n] is not specified, the current task execution is aborted.

**See Also:** [Section A.7.24, GET\\_TASK\\_INFO Built-In Procedure](#) , [Chapter 6, CONDITION HANDLERS](#)

**Example:** Refer to [Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#) for a detailed program example

## A.1.2 ABORT Condition

---

**Purpose:** Monitors the aborting of task execution

**Syntax:** ABORT <PROGRAM[n]>

**Details:**

- The ABORT condition is satisfied when the task is aborted. The actions specified by the condition handler will be performed.
- If PROGRAM [n] is not specified, the current task number is used.
- Actions that are routine calls will not be executed if task execution is aborted.
- The ABORT condition can be followed by the clause PROGRAM[n], where n is the task number to be monitored. Use GET\_TSK\_INFO to get a task number.

**See Also:** [Section A.3.40, CONDITION...ENDCONDITION Statement](#), [Section A.7.24, GET\\_TSK\\_INFO Built-In Procedure](#), [Chapter 6, CONDITION HANDLERS](#), [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

## A.1.3 ABORT Statement

---

**Purpose:** Terminates task execution and cancels any motion in progress (or pending)

**Syntax:** ABORT <PROGRAM[n]>

**Details:**

- After an ABORT, the program cannot be resumed. It must be restarted.
- The statement can be followed by the clause PROGRAM[n], where n is the task number to be aborted.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.1.4 ABORT\_TASK Built-In Procedure

**Purpose:** Aborts the specified running or paused task

**Syntax:** ABORT\_TASK(task\_name, force\_sw, cancel\_mtn\_sw, status)

Input/Output Parameters :

[in] task\_name : STRING

[in] force\_sw : BOOLEAN

[in] cancel\_mtn\_sw : BOOLEAN

[out] status : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- task\_name is the name of the task to be aborted. If task name is \*ALL\*, all executing or paused tasks are aborted except the tasks that have the ignore abort request attribute set.
- force\_sw, if true, specifies to abort a task even if the task has the ignore abort request set. force\_sw is ignored if task\_name is \*ALL\*.
- cancel\_mtn\_sw specifies whether motion is canceled for all groups belonging to the specified task.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.45, CONT\\_TASK Built-In Procedure](#) , [Section A.18.43, RUN\\_TASK Built-In Procedure](#) , [Section A.16.6, PAUSE\\_TASK Built-In Procedure](#) , [Section A.14.1, NOABORT Action](#) , [Section A.14.2, %NOABORT Translator Directive](#) , [Chapter 17, MULTI-TASKING](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#) for a detailed program example.

## A.1.5 ABS Built-In Function

**Purpose:** Returns the absolute value of the argument x, which can be an INTEGER or REAL expression

**Syntax:** ABS (x)

Function Return Type: INTEGER or REAL

Input/Output Parameters:

[in] x : INTEGER or REAL expression

%ENVIRONMENT Group : SYSTEM

**Details:**

Returns the absolute value of x, with the same data type as x.

## A.1.6 ACOS Built-In Function

**Purpose:** Returns the arc cosine (cos-1) in degrees of the specified argument

**Syntax:** ACOS (x)

Function Return Type : REAL

Input/Output Parameters :

[in] x : REAL

%ENVIRONMENT Group :SYSTEM

**Details:**

- x must be between -1.0 and 1.0; otherwise the program will abort with an error.
- Returns the arccosine of x.

**Example:** The following example sets ans\_r to the arccosine of -1 and writes this value to the screen. The output for the following example is 180 degrees.

```
routine take_acos
var
    ans_r: real
begin
    ans_r = acos (-1)
    WRITE ('acos -1 ', ans_r, CR)
END take_acos
```

**Figure A.1.6 (a) ACOS Built-In Function Example 1**

The second example causes the program to abort since the input value is less than -1 and not within the valid range.

```
routine take_acos
var
    ans_r: real
begin
    ans_r = acos (-1.5) -- causes program to abort
    WRITE ('acos -1.5 ', ans_r, CR)
END take_acos
```

**Figure A.1.6 (b) ACOS Built-In Function Example 2**

## A.1.7 ACT\_SCREEN Built-In Procedure

---

**Purpose:** Activates a screen

**Syntax:** ACT\_SCREEN

Input/Output Parameters:

[in] screen\_name : STRING

[out] old\_screen\_n : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Causes the display device associated with the screen to be cleared and all windows attached to the screen to be displayed.
- screen\_name must be a string containing the name of a previously defined screen, see **DEF\_SCREEN** Built-in.
- The name of the screen that this replaces is returned in old\_screen\_n .
- Requires the **USER** or **USER2** menu to be selected before activating the new screen, otherwise the status will be set to 9093.
  - To force the selection of the teach pendant user menu before activating the screen, use **FORCE\_SPMENU** (tp\_panel, SPI\_TPUSER, 1).
  - To force the selection of the CRT/KB user menu before activating the screen, use **FORCE\_SPMENU** (crt\_panel, SPI\_TPUSER, 1).
- If the **USER** menu is exited and re-entered, your screen will be reactivated as long as the KAREL task which called **ACT\_SCREEN** continues to run. When the KAREL task is aborted, the system's user screen will be re-activated. Refer to [Section 7.9, USER INTERFACE TIPS](#) for details on the system's user screen.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.4.8, DEF\\_SCREEN Built-In Procedure](#)

**Example:** Refer to the following section for detailed program examples:

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

## A.1.8 ACT\_TBL Built-In Procedure

**Purpose:** Acts on a key while displaying a table on the teach pendant

**Syntax:** ACT\_TBL(action, def\_item, table\_data, term\_char, attach\_wind, status)

Input/Output Parameters:

[in,out] action : INTEGER

[in,out] def\_item : INTEGER

[in,out] table\_data : XWORK\_T

[out] term\_char : INTEGER

[out] attach\_wind : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- The **INIT\_TBL** and **ACT\_TBL** built-in routines should only be used instead of **DISCTRL\_TBL** if special processing needs to be done with each keystroke or if function key processing needs to be done without exiting the table menu.
- The built-in **INIT\_TBL** must be called before using this built-in. Do not continue if an error occurs in **INIT\_TBL**.
- action must be one of the constants defined in the include file **KLEVKEYS.KL**. **ACT\_TBL** will act on the key and return. The following keys have special meanings:

- ky\_disp\_updt (Initial Display) - Table title is displayed, function key labels are displayed. Default item is displayed and highlighted in first line. Remaining lines are displayed. Dynamic display is initiated for all values. This should be the first key passed into ACT\_TBL.
- ky\_reissue (Read Key) - ACT\_TBL reads a key, acts on it, and returns it in action.
- ky\_cancel (Cancel Table) - All dynamic display is canceled. This should be the last key passed into ACT\_TBL if term\_char does not equal ky\_new\_menu. If a new menu was selected, ACT\_TBL will have already canceled the dynamic display and you should not use ky\_cancel.
- def\_item is the row containing the item you want to be highlighted when the table is entered. On return, def\_item is the row containing the item that was currently highlighted when the termination character was pressed.
- table\_data is used to display and control the table. Do not change this data; it is used internally.
- term\_char receives a code indicating the character or other condition that terminated the table. The codes for key terminating conditions are defined in the include file KLEVKEYS.KL. Keys normally returned are pre-defined constants as follows:
  - ky\_undef - No termination character was pressed
  - ky\_select - A selectable item was selected
  - key\_new\_menu - A new menu was selected
  - ky\_f1 through ky\_f10 - A function key was selected
- attach\_wind should be set TRUE if the table manager needs to be attached to the display devices when action is ky\_disp\_updt and detached from the display devices when action is ky\_cancel. If it is already attached, this parameter can be set to FALSE. Typically this should be TRUE.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Refer to the example for the INIT\_TBL built-in, [Section A.9.12, INIT\\_TBL Built-In Procedure](#).

## A.1.9 ADD\_BYNAMEPC Built-In Procedure

---

**Purpose:** To add an integer, real, or string value into a KAREL byte given a data buffer.

**Syntax:** ADD\_BYNAMEPC(dat\_buffer, dat\_index, prog\_name, var\_name, status)

Input/Output Parameters:

[in] dat\_buffer : ARRAY OF BYTE

[in,out] dat\_index : INTEGER

[in] prog\_name : STRING

[in] var\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PC

**Details:**

- dat\_buffer - an array of up to 244 bytes.
- dat\_index - the starting byte number to place the string value.
- prog\_name - specifies the name of the program that contains the specified variable.
- var\_name - refers to a static program variable. This is only supported by an integer, real, or string variable (arrays and structures are not supported).

- status - the status of the attempted operation. If not 0, then an error occurred and data was not placed into the buffer.

The ADD\_BYNAMEPC built-in adds integer, real, and string values to the data buffer in the same manner as the KAREL built-ins ADD\_INTPC, ADD\_REALPC, and ADD\_STRINGPC.

**See Also:** [Section A.1.9, ADD\\_BYNAMEPC Built-In Procedure](#), [Section A.1.11, ADD\\_INTPC Built-In Procedure](#), [Section A.1.12, ADD\\_REALPC Built-In Procedure](#), [Section A.1.13, ADD\\_STRINGPC Built-In Procedure](#)

**Example:** See the following for an example of the ADD\_BYNAMEPC built-in.

```
PROGRAM TESTBYNM
%ENVIRONMENT PC
CONST
    er_abort = 2
VAR
    dat_buffer: ARRAY[100] OF BYTE
    index:      INTEGER
    status:     INTEGER
BEGIN
    index = 1
    ADD_BYNAMEPC(dat_buffer, index, 'TESTDATA', 'INDEX', status)
    IF status<>0 THEN
        POST_ERR(status, '', 0, er_abort)
    ENDIF
END testbyn
```

**Figure A.1.9 ADD\_BYNAMEPC Built-In Procedure**

## A.1.10 ADD\_DICT Built-In Procedure

**Purpose:** Adds the specified dictionary to the specified language.

**Syntax:** ADD\_DICT(file\_name, dict\_name, lang\_name, add\_option, status)

Input/Output Parameters:

- [in] file\_name : STRING
- [in] dict\_name : STRING
- [in] lang\_name : STRING
- [in] add\_option : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- file\_name specifies the device, path, and file name of the dictionary file to add. The file type is assumed to be .TX (text file).
- dict\_name specifies the name of the dictionary to use when reading and writing dictionary elements. Only 4 characters are used.
- lang\_name specifies to which language the dictionary will be added. One of the following pre-defined constants should be used:

- dp\_default
  - dp\_english
  - dp\_japanese
  - dp\_french
  - dp\_german
  - dp\_spanish
- The default language should be used unless more than one language is required.
  - add\_option should be the following:
    - dp\_dram Dictionary will be loaded to DRAM memory and retained until the next INIT START.
  - status explains the status of the attempted operation. If not equal to 0, then an error occurred adding the dictionary file.

**See Also:** [Section A.18.2, READ\\_DICT Built-In Procedure](#), [Section A.23.6, WRITE\\_DICT Built-In Procedure](#), [Section A.18.9, REMOVE\\_DICT Built-In Procedure](#), [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.11.1, Dictionary Files \(DCALPHEG.UTX\)](#)

## A.1.11 ADD\_INTPC Built-In Procedure

---

**Purpose:** To add an INTEGER value (type 16 - 10 HEX) into a KAREL byte data buffer.

**Syntax:** ADD\_INTPC(dat\_buffer, dat\_index, number, status)

Input/Output Parameters:

[in] dat\_buffer : ARRAY OF BYTE

[in,out] dat\_index : INTEGER

[in] number : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PC

### Details:

- dat\_buffer - an array of up to 244 bytes.
- dat\_index - the starting byte number to place the integer value.
- number - the integer value to place into the buffer.
- status - the status of the attempted operation. If not 0, then an error occurred and data was not put into the buffer.

The KAREL built-ins ADD\_BYNAMEPC, ADD\_INTPC, ADD\_REALPC, and ADD\_STRINGPC can be used to format a KAREL byte buffer in the following way. INTEGER data is added to the buffer as follows (buffer bytes are displayed in HEX):

- beginning index = dat\_index
- 2 bytes - variable type
- 4 bytes - the number
- 2 bytes of zero (0) - end of buffer marker

The following is an example of an INTEGER placed into a KAREL array of bytes starting at index = 1:

```
0 10 0 0 0 5 0 0
```

where:

0 10 = INTEGER variable type

0 0 0 5 = INTEGER number 5

0 0 = end of data in the buffer

On return from the built-in, index = 7.

**See Also:** [Section A.1.9, ADD\\_BYNAMEPC Built-In Procedure](#) , [Section A.1.11, ADD\\_INTPC Built-In Procedure](#) , [Section A.1.12, ADD\\_REALPC Built-In Procedure](#) , [Section A.1.13, ADD\\_STRINGPC Built-In Procedure](#)

**Example:** Refer to the TESTDATA example in the [Section A.19.7, SEND\\_DATAPC Built-In Procedure](#) .

## A.1.12 ADD\_REALPC Built-In Procedure

**Purpose:** To add a REAL value (type 17 - 11 HEX) into a KAREL byte data buffer.

**Syntax:** ADD\_REALPC(dat\_buffer, dat\_index, number, status)

Input/Output Parameters:

[in] dat\_buffer : ARRAY OF BYTE

[in,out] dat\_index : INTEGER

[in] number : REAL

[out] status : INTEGER

%ENVIRONMENT Group : PC

### Details:

- dat\_buffer - an array of up to 244 bytes.
- dat\_index - the starting byte number to place the real value.
- number - the real value to place into the buffer.
- status - the status of the attempted operation. If not 0, then an error occurred and data was not placed into the buffer.

The KAREL built-ins ADD\_BYNAMEPC, ADD\_INTPC, ADD\_REALPC, and ADD\_STRINGPC can be used to format a KAREL byte buffer in the following way:

REAL data is added to the buffer as follows (buffer bytes are displayed in HEX):

- beginning index = dat\_index
- 2 bytes - variable type
- 4 bytes - the number
- 2 bytes of zero (0) - end of buffer marker

The following is an example of an REAL placed into a KAREL array of bytes starting at index = 1:

```
0 11 43 AC CC CD 0 0
```

where:

0 D1 = REAL variable type

43 AC CC CD = real number 345.600006

0 0 = end of data in the buffer

On return from the built-in, index = 7.

**See Also:** [Section A.1.9, ADD\\_BYNAMEPC Built-In Procedure](#), [Section A.1.11, ADD\\_INTPC Built-In Procedure](#), [Section A.1.13, ADD\\_STRINGPC Built-In Procedure](#)

**Example:** Refer to the TESTDATA example in the [Section A.19.7, SEND\\_DATAPC Built-In Procedure](#).

## A.1.13 ADD\_STRINGPC Built-In Procedure

---

**Purpose:** To add a string value (type 209 - D1 HEX) into a KAREL byte data buffer.

**Syntax:** ADD\_STRINGPC(dat\_buffer, dat\_index, item, status)

Input/Output Parameters:

[in] dat\_buffer : ARRAY OF BYTE

[in,out] dat\_index : INTEGER

[in] item : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PC

**Details:**

- dat\_buffer - an array of up to 244 bytes.
- dat\_index - the starting byte number to place the string value.
- item - the string value to place into the buffer.
- status - the status of the attempted operation. If not 0, then an error occurred and data was not placed into the buffer.

The KAREL built-ins ADD\_BYNAMEPC, ADD\_INTPC, ADD\_REALPC, and ADD\_STRINGPC can be used to format a KAREL byte buffer in the following way:

STRING data is added to the buffer as follows:

- beginning index = dat\_index
- 2 bytes - variable type
- 1 byte - length of text string
- text bytes
- 2 bytes of zero (0) - end of buffer marker

The following is an example of an STRING placed into a KAREL array of bytes starting at index = 1:

```
0 D1 7 4D 48 53 48 45 4C 4C 0 0 0
```

where:

0 D1 = STRING variable type

7 = there are 7 characters in string MHSHELL

4D 48 53 48 45 4C 4C 0 = MHSHELL with end of string 0

0 0 = end of data in the buffer

On return from the built-in, index = 12.

**See Also:** [Section A.1.9, ADD\\_BYNAMEPC Built-In Procedure](#) , [Section A.1.11, ADD\\_INTPC Built-In Procedure](#) , [Section A.1.12, ADD\\_REALPC Built-In Procedure](#) , [Section A.1.13, ADD\\_STRINGPC Built-In Procedure](#)

**Example:** Refer to the TESTDATA example in the [Section A.19.7, SEND\\_DATAPC Built-In Procedure](#) .

## A.1.14 %ALPHABETIZE Translator Directive

**Purpose:** Specifies that static variables will be created in alphabetical order when p-code is loaded.

**Syntax:** %ALPHABETIZE

**Details:**

- Static variables can be declared in any order in a KAREL program and %ALPHABETIZE will cause them to be displayed in alphabetical order in the DATA menu or KCL> SHOW VARS listing.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

## A.1.15 APPEND\_NODE Built-In Procedure

**Purpose:** Adds an uninitialized node to the end of the PATH argument

**Syntax:** APPEND\_NODE(path\_var, status)

Input/Output Parameters:

[in] path\_var : PATH

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- path\_var is the path variable to which the node is appended.
- The appended PATH node is uninitialized. The node can be assigned values by directly referencing its NODEDATA structure.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.4.13, DELETE\\_NODE Built-In Procedure](#)

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.1.16 APPEND\_QUEUE Built-In Procedure

---

**Purpose:** Appends an entry to a queue if the queue is not full

**Syntax:** APPEND\_QUEUE(value, queue, queue\_data, sequence\_no, status)

Input/Output Parameters:

[in] value : INTEGER

[in,out] queue : QUEUE\_TYPE

[in,out] queue\_data : ARRAY OF INTEGER

[out] sequence\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBQMGR

**Details:**

- value specifies the value to be appended to the queue.
- queue specifies the queue variable for the queue.
- queue\_data specifies the array used to hold the data in the queue. The length of this array determines the maximum number of entries in the queue.
- sequence\_no is returned with the sequence number of the entry just appended.
- status is returned with the zero if an entry can be appended to the queue. Otherwise it is returned with **61001**, Queue is full.

**See Also:** [Section A.4.14, DELETE\\_QUEUE Built-In Procedure](#) , [Section A.9.15, INSERT\\_QUEUE Built-In Procedure](#) . Refer to [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#) , for more information and an example.

## A.1.17 APPROACH Built-In Function

---

**Purpose:** Returns a unit VECTOR representing the z-axis of a POSITION argument

**Syntax:** APPROACH(posn)

Function Return Type: VECTOR

Input/Output Parameters:

[in] posn : POSITION

%ENVIRONMENT Group : VECTR

**Details:**

- Returns a VECTOR consisting of the approach vector (positive z-axis) of the argument posn.

**Example:** This program allows you to create a position that is 500 mm away from another position along the z-axis.

```
PROGRAM p_approach
VAR
    start_pos : POSITION
    app_vector : VECTOR
BEGIN
    app_vector = APPROACH (start_pos) --sets app_vector equal to the
                                      --z-axis of start_pos
    start_pos.location = start_pos.location + app_vector *500
                                      --creates start_pos + 500 mm
                                      --in z direction
END p_approach
```

**Figure A.1.17 APPROACH Function**

**NOTE**

APPROACH has been left in for older versions of KAREL. You should now directly access the vectors of a POSITION (for example, posn. approach.)

## A.1.18 ARRAY Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as ARRAY data type

**Syntax:** ARRAY<[size{,size}]> OF data\_type

where:

size : an INTEGER literal or constant

data\_type : any type except PATH

**Details:**

- size indicates the number of elements in an ARRAY variable.
- size must be in the range 1 through 32767 and must be specified in a normal ARRAY variable declaration. The amount of available memory in your controller might restrict the maximum size of an ARRAY.
- Individual elements are referenced by the ARRAY name and the subscript size. For example, table[1] refers to the first element in the ARRAY table.
- An entire ARRAY can be used only in assignment statements or as an argument in routine calls. In an assignment statement, both ARRAY variables must be of the same size and data\_type. If size is different, the program will be translated successfully but will be aborted during execution, with error **12304, Array Length Mismatch**.
- size is not specified when declaring ARRAY routine parameters; an ARRAY of any size can be passed as an ARRAY parameter to a routine.
- size is not used when declaring an ARRAY return type for a function. However, the returned ARRAY must be of the same size as the ARRAY to which it is assigned in the function call.
- Each element is of the same type designated by data\_type.
- Valid ARRAY operators correspond to the valid operators of the individual elements in the ARRAY.
- Individual elements of an array can be read or written only in the format that corresponds to the data type of the ARRAY.

- Arrays of multiple dimensions can be defined. Refer to [Chapter 2, LANGUAGE ELEMENTS](#) for more information.
- Variable-sized arrays can be defined. Refer to [Chapter 2, LANGUAGE ELEMENTS](#) for more information.

**See Also:** [Section A.1.19, ARRAY\\_LEN Built-In Function](#), [Chapter 5, ROUTINES](#), for information on passing ARRAY variables as arguments in routine calls, [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.1.19 ARRAY\_LEN Built-In Function

---

**Purpose:** Returns the number of elements contained in the specified array argument

**Syntax:** `ARRAY_LEN(ary_var)`

Function Return Type : INTEGER

Input/Output Parameters:

[in] `ary_var :ARRAY`

%ENVIRONMENT Group :SYSTEM

- The returned value is the number of elements declared for `ary_var`, not the number of elements that have been initialized in `ary_var`.

**Example:** Refer to [Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#), for a detailed program example.

## A.1.20 ASIN Built-In Function

---

**Purpose:** Returns arcsine (sin<sup>-1</sup>) in degrees of the specified argument

**Syntax:** `ASIN(x)`

Function Return Type: REAL

Input/Output Parameters:

[in] `x : REAL`

%ENVIRONMENT Group : SYSTEM

### Details:

- Returns the arcsine of `x`.
- `x` must be between -1 and 1, otherwise the program will abort with an error.

**Example:** The following example sets ans\_r to the arcsine of -1 and writes this value to the screen. The output for the following example is -90 degrees.

```
ROUTINE take_asin
VAR
    ans_r: REAL
BEGIN
    ans_r = ASIN (-1)
    WRITE ('asin -1 ', ans_r, CR)
END take_asin
```

**Figure A.1.20 (a) ASIN Built-In Function Example 1**

The second example causes the program to abort since the input value is less than -1 and not within the valid range.

```
ROUTINE take_asin
VAR
    ans_r: REAL
BEGIN
    ans_r = ASIN (-1.5) -- causes program to abort
    WRITE ('asin -1.5 ', ans_r, CR)
END take_asin
```

**Figure A.1.20 (b) ASIN Built-In Function Example 2**

## A.1.21 Assignment Action

**Purpose:** Sets the value of a variable to the result of an evaluated expression

**Syntax:** variable {[subscript{,subscript}]} . field} = expn

where:

variable : any KAREL variable

subscript : an INTEGER expression

expn : a valid KAREL expression

field : any field from a structured variable

### Details:

- variable can be any user-defined variable, system variable with write access, or output port array with write access.
- subscript is used to access elements of an array.
- field is used to access fields in a structure.
- expn must be of the same type as the variable or element of variable.
- An exception is that an INTEGER expression can be assigned to a REAL. Any positional types can be assigned to each other.
- Only system variables with write access (listed as RW) can be used on the left side of an assignment statement. System variables with read only (RO) or read write (RW) access can be used on the right side.
- Input port arrays cannot be used on the left side of an assignment statement.

**See Also:** [Chapter 3, USE OF OPERATORS](#), for detailed information about expressions and their evaluation, [Chapter 2, LANGUAGE ELEMENTS](#), for more information about using assignment actions.

**Example:** The following example uses the assignment action to turn DOUT[1] off and set port\_var equal to DOUT[2] when EVENT[1] turns on.

```
CONDITION [1] :
  WHEN EVENT[1] DO
    DOUT[1] = OFF
    port_var = DOUT[2]
  ENDCONDITION
```

**Figure A.1.21 Assignment Action**

## A.1.22 Assignment Statement

---

**Purpose:** Sets the value of a variable to the result of an evaluated expression

**Syntax:** variable {[subscript{,subscript}]} . field} = expn

where:

variable : any KAREL variable

subscript : an INTEGER expression

expn : a valid KAREL expression

field : any field from a structured variable

### Details:

- variable can be any user-defined variable, system variable with write access, or output port array with write access.
- subscript is used to access elements of an array.
- field is used to access fields in a structure.
- expn must be of the same type as the variable or element of variable.
- An exception is that an INTEGER expression can be assigned to a REAL. Any positional types can be assigned to each other. INTEGER, SHORT, and BYTE can be assigned to each other.
- If variable is of type ARRAY, and no subscript is supplied, the expression must be an ARRAY of the same type and size. A type mismatch will be detected during translation. A size mismatch will be detected during execution and causes the program to abort with error **12304, Array Length Mismatch**.
- If variable is a user-defined structure, and no field is supplied, the expression must be a structure of the same type.
- Only system variables with write access (listed as RW) can be used on the left side of an assignment statement. System variables with read only (RO) or read write (RW) access can be used on the right side.

If read only system variables are passed as parameters to a routine, they are passed by value, so any attempt to modify them (with an assignment statement) through the parameter in the routine has no effect.

- Input port arrays cannot be used on the left side of an assignment statement.

**See Also:** [Chapter 3, USE OF OPERATORS](#) , for detailed information about expressions and their evaluation, [Chapter 2, LANGUAGE ELEMENTS](#) . Refer to [Appendix B, KAREL EXAMPLE PROGRAMS](#) for more detailed program examples.

**Example:** The following example assigns an INTEGER literal to an INTEGER variable and then increments that variable by a literal and value.

```
int_var = 5
int_var = 5 + int_var
```

**Figure A.1.22 (a) Assignment Statement Example 1**

The next example multiplies the system variable `$SPEED` by a REAL value. It is then used to assign the ARRAY variable `array_1`, element `loop_count` to the new value of the system variable `$SPEED`.

```
$SPEED = $SPEED * .25
array_1[loop_count] = $SPEED
```

**Figure A.1.22 (b) Assignment Statement Example 2**

The last example assigns all the elements of the ARRAY `array_1` to those of ARRAY `array_2`, and all the fields of structure `struc_var_1` to those of `struc_var_2`.

```
array_2 = array_1
struc_var_2 = struc_var_1
```

**Figure A.1.22 (c) Assignment Statement Example 3**

## A.1.23 ATAN2 Built-In Function

**Purpose:** Returns a REAL angle, measured counterclockwise in degrees, from the positive x-axis to a line connecting the origin and a point whose x- and y- coordinates are specified as the x- and y- arguments

**Syntax:** `ATAN2(x1, y1)`

Function Return Type : REAL

Input/Output Parameters :

[in] `x1` : REAL

[in] `y1` : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- `x1` and `y1` specify the x and y coordinates of the point.
- If `x1` and `y1` are both zero, the interpreter will abort the program.

**Example:** The following example uses the values 100, 200, and 300 respectively for x, y, and z to compute the orientation component direction. The position, `p1` is then defined to be a position with direction as its orientation component.

```
PROGRAM p_atan2
VAR
    p1 : POSITION
```

```

x, y, z, direction : REAL
BEGIN
  x = 100 -- use appropriate values
  y = 200 -- for x,y,z on
  z = 300 -- your robot
  direction = ATAN2(x, y)
  p1 = POS(x, y, z, 0, 0, direction, 'n') --r orientation component
                                              --of POS equals angle
                                              --returned by
END p_atan2
ATAN2(100,200)

```

**Figure A.1.23 ATAN2 Built-In Function**

## A.1.24 ATTACH Statement

---

**Purpose:** Gives the KAREL program control of motion for the robot arm and auxiliary and extended axes

**Syntax:** ATTACH

**Details:**

- Used with the RELEASE statement. If motion control is not currently released from program control, the ATTACH statement has no effect.
- If the teach pendant is still enabled, execution of the KAREL program is delayed until the teach pendant is disabled. The task status will show a hold of **attach done**.
- Stopped motions can only be resumed following execution of the ATTACH statement.

**See Also:** [Section A.18.8, RELEASE Statement](#), [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#), for a detailed program example.

## A.1.25 ATT\_WINDOW\_D Built-In Procedure

---

**Purpose:** Attach a window to the screen on a display device

**Syntax:** ATT\_WINDOW\_D(window\_name, disp\_dev\_nam, row, col, screen\_name, status)

**Input/Output Parameters:**

[in] window\_name : STRING

[in] disp\_dev\_nam : STRING

[in] row : INTEGER

[in] col : INTEGER

[out] screen\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Causes data in the specified window to be displayed or attached to the screen currently active on the specified display device.
- window\_name must be a previously defined window.
- disp\_dev\_nam must be one of the display devices already defined:
  - 'CRT' – CRT Device
  - 'TP' – Teach Pendant Device
- row and col indicate the position in the screen. Row 1 indicates the top row; col 1 indicates the left-most column. The entire window must be visible in the screen where positioned. For example, if the screen is 24 rows by 80 columns (as defined by its associated display device) and the window is 2 rows by 80 columns, row must be in the range 1-23; col must be 1.
- The name of the active screen is returned in screen\_name. This can be used to detach the window later.
- It is an error if the window is already attached to the screen.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

## A.1.26 ATT\_WINDOW\_S Built-In Procedure

**Purpose:** Attach a window to a screen

**Syntax:** ATT\_WINDOW\_S(window\_name, screen\_name, row, col, status)

Input/Output Parameters:

[in] window\_name : STRING

[in] screen\_name : STRING

[in] row : INTEGER

[in] col : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Causes data in the specified window to be displayed or attached to the specified screen at a specified row and column.
- window\_name and screen\_name must be previously defined window and screen names.
- row and col indicate the position in the screen. Row 1 indicates the top row; col 1 indicates the left-most column. The entire window must be visible in the screen as positioned. For example, if the screen is 24 rows by 80 columns (as defined by its associated display device) and the window is 2 rows by 80 columns, row must be in the range 1-23; col must be 1.
- If the screen is currently active, the data will immediately be displayed on the device. Otherwise, there is no change in the displayed data.
- It is an error if the window is already attached to the screen.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section 7.9, USER INTERFACE TIPS](#) , [Section A.4.16, DET\\_WINDOW Built-In Procedure](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.1.27 AVL\_POS\_NUM Built-In Procedure

---

**Purpose:** Returns the first available position number in a teach pendant program

**Syntax:** AVL\_POS\_NUM(open\_id, pos\_num, status)

Input/Output Parameters:

[in] open\_id : INTEGER

[out] pos\_num : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : TPE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- pos\_num is set to the first available position number.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.2 - B - KAREL LANGUAGE DESCRIPTION

---

### A.2.1 BOOLEAN Data Type

---

**Purpose:** Defines a variable, function return type, or routine parameter as a BOOLEAN data type

**Syntax:** BOOLEAN

**Details:**

- The BOOLEAN data type represents the BOOLEAN predefined constants TRUE, FALSE, ON, and OFF.

[Table A.2.1](#) lists some examples of valid and invalid BOOLEAN values used to represent the Boolean predefined constants.

**Table A.2.1 Valid and Invalid BOOLEAN Values**

VALID	INVALID	REASON
TRUE	T	Must use entire word
ON	1	Cannot use INTEGER values

- TRUE and FALSE typically represent logical flags, and ON and OFF typically represent signal states. TRUE and ON are equivalent, as are FALSE and OFF.
- Valid BOOLEAN operators are
  - AND, OR, and NOT
  - Relational operators (>, >=, =, <>, <, and <=)
- The following have BOOLEAN values:

- BOOLEAN constants, whether predefined or user-defined (for example, ON is a predefined constant)
- BOOLEAN variables and BOOLEAN fields in a structure
- ARRAY OF BOOLEAN elements
- Values returned by BOOLEAN functions, whether user-defined or built-in (for example, IN\_RANGE(pos\_var))
- Values resulting from expressions that use relational or BOOLEAN operators (for example, x > 5.0)
- Values of digital ports (for example, DIN[2])
- Only BOOLEAN expressions can be assigned to BOOLEAN variables, returned from BOOLEAN function routines, or passed as arguments to BOOLEAN parameters.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.2.2 BYNAME Built-In Function

**Purpose:** Allows a KAREL program to pass a variable, whose name is contained in a STRING, as a parameter to a KAREL routine. This means the programmer does not have to determine the variable name during program creation and translation.

**Syntax:** BYNAME (prog\_name, var\_name, entry)

Input/Output Parameters:

[in] prog\_name : STRING

[in] var\_name : STRING

[in,out] entry : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- This built-in can be used only to pass a parameter to a KAREL routine.
- entry returns the entry number in the variable data table where var\_name is located. This variable does not need to be initialized and should not be modified.
- prog\_name specifies the name of the program that contains the specified variable. If prog\_name is equal to " (double quotes), then the routine defaults to the task name being executed.

- var\_name must refer to a static, program variable.
- If var\_name does not contain a valid variable name or if the variable is not of the type expected as a routine parameter, the program is aborted.
- System variables cannot be passed using BYNAME.
- The PATH data type cannot be passed using BYNAME. However, a user-defined type that is a PATH can be used instead.

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.2.3 BYTE Data Type

---

**Purpose:** Defines a variable as a BYTE data type

**Syntax:** BYTE

**Details:**

- BYTE has a range of ( $0 \leq n \geq 255$ ). No uninitialized checking is done on bytes.
- BYTES are allowed only within an array or within a structure.
- BYTES can be assigned to SHORTs and INTEGERs, and SHORTs and INTEGERs can be assigned to BYTES. An assigned value outside the BYTE range will be detected during execution and cause the program to abort.

**Example:** The following example defines an array of BYTE and a structure containing BYTES.

```
PROGRAM byte_ex
%NOLOCKGROUP
    TYPE
        mystruct = STRUCTURE
            param1: BYTE
            param2: BYTE
            param3: SHORT
        ENDSTRUCTURE
    VAR
        array_byte: ARRAY[10] OF BYTE
        myvar: mystruct
    BEGIN
        array_byte[1] = 254
        myvar.param1 = array_byte[1]
    END byte_ex
```

Figure A.2.3 BYTE Data Type

## A.2.4 BYTES\_AHEAD Built-In Procedure

---

**Purpose:** Returns the number of bytes of input data presently in the read-ahead buffer for a KAREL file. Allows KAREL programs to check instantly if data has been received from a serial port and is available to be read by the program. BYTES\_AHEAD is also supported on socket messaging and pipes.

**Syntax:** BYTES\_AHEAD(file\_id, n\_bytes, status)

Input/Output Parameters:

```
[in] file_id : FILE
[out] n_bytes : INTEGER
[out] status : INTEGER
%ENVIRONMENT Group : FLBT
```

**Details:**

- file\_id specifies the file that was opened.
- The file\_id must be opened with the ATR\_READAHD attribute set greater than zero.
- n\_bytes is the number of bytes in the read\_ahead buffer.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- A non-zero status will be returned for non-serial devices such as files.

**See Also:** [Section 7.2.1, Setting File and Port Attributes](#)

**Example:** The following example will clear Port 2 (FLPY:) from any bytes still remaining to be read.

```
ROUTINE purge_port
VAR
    s1          : STRING[1]
    n_try       : INTEGER
    n_bytes     : INTEGER
    stat        : INTEGER
BEGIN
    stat=SET_PORT_ATR (port_2, ATR_READAHD, 1) --sets FLPY: to have a
                                                --read ahead buffer of 128 bytes
    OPEN FILE fi('RO', 'rdahd.tst')
    REPEAT
        BYTES_AHEAD (fi, n_bytes, stat)
                    --Get number of bytes ready
                    --to be read
        if (n_bytes = 0) then      --if there are no bytes then set stat
            stat = 282
        endif
        if (n_bytes >= 1) then    --there are bytes to be read
            read fi(s1::1)        --read in one byte at a time
            stat=io_status (fi)   --get the status of the read operation
        endif
        UNTIL stat <> 0          --continue until no more bytes are left
    END purge_port
    BEGIN
        -- main program text here
    END bytes_ahd
```

**Figure A.2.4 BYTES\_AHEAD Built-In Procedure**

## A.2.5 BYTES\_LEFT Built-In Function

**Purpose:** Returns the number of bytes remaining in the current input data record

**Syntax:** BYTES\_LEFT(file\_id)

Function Return Type: INTEGER

Input/Output Parameters:

[in] file\_id : FILE

%ENVIRONMENT Group : FLBT

**Details:**

- file\_id specifies the file that was opened.
- If no read or write operations have been done or the last operation was a READ file\_id (CR), a zero is returned.
- If file\_id does not correspond to an opened file or one of the pre-defined files opened to the respective CRT/KB, teach pendant, and vision windows, the program is aborted.

**NOTE**

An infeed character (LF) is created when the **ENTER** key is pressed, and is counted by **BYTES\_LEFT**.

- This function will return a non-zero value only when data is input from a keyboard (teach pendant or CRT/KB), not from files or ports.

**⚠ WARNING**

This function is used exclusively for reading from a window to determine if more data has been entered. Do not use this function with any other file device. Otherwise, you could injure personnel or damage equipment.

**See Also:** [Section 7.9.1, USER Menu on the Teach Pendant](#), [Section 7.9.2, USER Menu on the CRT/KB](#)

**Example:** The following example reads the first number, rqd\_field , and then uses **BYTES\_LEFT** to determine if the user entered any additional numbers. If so, these numbers are then read.

```
PROGRAM p_bytesleft
%NOLOCKGROUP
%ENVIRONMENT flbt
CONST
    default_1 = 0
    default_2 = -1
VAR rqd_field, opt_field_1, opt_field_2: INTEGER
BEGIN
    WRITE('Enter integer field(s): ')
    READ(rqd_field)
    IF BYTES_LEFT(TPDISPLAY) > 0 THEN
        READ(opt_field_1)
    ELSE
        opt_field_1 = default_1
    ENDIF
    IF BYTES_LEFT(TPDISPLAY) > 0 THEN
        READ(opt_field_2)
    ELSE
        opt_field_2 = default_2
    ENDIF
END p_bytesleft
```

**Figure A.2.5 BYTES\_LEFT Built-In Function**

## A.3 - C - KAREL LANGUAGE DESCRIPTION

---

### A.3.1 CALL\_PROG Built-In Procedure

---

**Purpose:** Allows a KAREL program to call an external KAREL or teach pendant program. This means that the programmer does not have to determine the program to be called until run time.

**Syntax:** CALL\_PROG(prog\_name, prog\_index)

Input/Output Parameters:

[in] prog\_name : STRING

[in,out] prog\_index : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- prog\_name is the name of the program to be executed, in the current calling task.
- prog\_index returns the entry number in the program table where prog\_name is located. This variable does not need to be initialized and should not be modified.
- CALL\_PROG cannot be used to run internal or external routines.

**See Also:** [Section A.3.57, CURR\\_PROG Built-In Function](#) and [Section A.3.2, CALL\\_PROGLIN Built-In Procedure](#)

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

### A.3.2 CALL\_PROGLIN Built-In Procedure

---

**Purpose:** Allows a KAREL program to call an external KAREL or teach pendant program, beginning at a specified line. This means that the programmer does not need to know, at creation and translation, what program will be called. The programmer can decide this at run time.

**Syntax:** CALL\_PROGLIN(prog\_name, prog\_line, prog\_index, pause\_entry)

Input/Output Parameters:

[in] prog\_name : STRING

[in] prog\_line : INTEGER

[in,out] prog\_index : INTEGER

[in] pause\_entry : BOOLEAN

%ENVIRONMENT Group : BYNAM

**Details:**

- prog\_name is the name of the program to be executed, in the current calling task.
- prog\_line specifies the line at which to begin execution for a teach pendant program. 0 or 1 is used for the beginning of the program.
- KAREL programs always execute at the beginning of the program.

- prog\_index returns the entry number in the program table where prog\_name is located. This variable does not need to be initialized and should not be modified.
- pause\_entry specifies whether to pause program execution upon entry of the program.
- CALL\_PROGLIN cannot be used to run internal or external routines.

**See Also:** [Section A.3.57, CURR\\_PROG Built-In Function](#) , [Section A.3.1, CALL\\_PROG Built-In Procedure](#)

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## A.3.3 CANCEL Action

---

**Purpose:** Terminates any motion in progress

**Syntax:** CANCEL <GROUP[n{,n}]>

**Details:**

- Cancels a motion currently in progress or pending (but not stopped) for one or more groups.
- CANCEL does not cancel motions that are already stopped. To cancel a motion that is already stopped, use the CNCL\_STP\_MTN built-in routine.
- If the group clause is not present, all groups for which the task has control (when the condition is defined) will be canceled. In particular, if the program containing the condition handler definition contains the %NOLOCKGROUP directive, the CANCEL action will not cancel motion in any group.
- If a motion that is canceled and is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are canceled.
- The robot and auxiliary or extended axes decelerate smoothly to a stop. The remainder of the motion is canceled.
- Canceled motions are treated as completed and cannot be resumed.
- The CANCEL action in a global condition handler cancels any pending motions.

## A.3.4 CANCEL Statement

---

**Purpose:** Terminates any motion in progress.

**Syntax:** CANCEL <GROUP[n{,n}]>

**Details:**

- Cancels a motion currently in progress or pending (but not stopped) for one or more groups.
- CANCEL does not cancel motions that are already stopped. To cancel a motion that is already stopped, use the CNCL\_STP\_MTN built-in routine.
- If the group clause is not present, all groups for which the task has control will be canceled. In particular, if the program using the CANCEL statement contains the %NOLOCKGROUP directive, the CANCEL statement will not cancel motion in any group.
- If a motion that is canceled is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are canceled.
- The robot and auxiliary axes decelerate smoothly to a stop. The remainder of the motion is canceled.
- Canceled motions are treated as completed and cannot be resumed.
- CANCEL does not affect stopped motions. Stopped motions can be resumed.

- If an interrupt routine executes a CANCEL statement and the interrupted statement was a motion statement, when the interrupted program resumes, execution normally resumes with the statement following the motion statement.
- CANCEL might not work as expected if it is used in a routine called by a condition handler. The motion might already be put on the stopped motion queue before the routine is called. Use a CANCEL action directly in the condition handler to be sure the motion is canceled.
- Motion cannot be canceled for a different task.

See Also: [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

## A.3.5 CANCEL FILE Statement

**Purpose:** Cancels a READ or WRITE statement that is in progress.

**Syntax:** CANCEL FILE [file\_var]

where:

file\_var : a FILE variable

**Details:**

- Used to cancel input or output on a specified file
- The built-in function `IO_STATUS` can be used to determine if a CANCEL FILE operation was successful or, if it failed to determine the reason for the failure.

See Also: [Section A.9.19, IO\\_STATUS Built-In Function](#), [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#), [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** The following example reads an integer, but cancels the read if the F1 key is pressed.

```
PROGRAM can_file_ex
%ENVIRONMENT FLBT
%ENVIRONMENT UIF
%NOLOCKGROUP
VAR
    int_var: INTEGER
ROUTINE cancel_read
BEGIN
    CANCEL FILE TPDISPLAY
END cancel_read
BEGIN
CONDITION[1]:
    WHEN TPIN[ky_f1]+ DO
        cancel_read
    ENABLE CONDITION[1]
ENDCONDITION

ENABLE CONDITION[1]
REPEAT
    -- Read an integer, but cancel if F1 pressed
    CLR_IO_STAT(TPDISPLAY)
    WRITE(CR, 'Enter an integer: ')
    READ(int_var)
```

```
UNTIL FALSE
end can_file_ex
```

**Figure A.3.5 CANCEL FILE Statement**

## A.3.6 CHECK\_DICT Built-In Procedure

---

**Purpose:** Checks the specified dictionary for a specified element

**Syntax:** CHECK\_DICT(dict\_name, element\_no, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] element\_no : STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- dict\_name is the name of the dictionary to check.
- element\_no is the element number within the dictionary.
- status explains the status of the attempted operation. If not equal to 0, then the element could not be found.

**See Also:** [Section A.1.10, ADD\\_DICT Built-In Procedure](#) , [Section A.18.2, READ\\_DICT Built-In Procedure](#) , [Section A.23.6, WRITE\\_DICT Built-In Procedure](#) , [Section A.18.9, REMOVE\\_DICT Built-In Procedure](#) . Refer to the program example for the [Section A.4.22, DISCTRL\\_LIST Built-In Procedure](#) , Chapter 11, DICTIONARIES AND FORMS .

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.3.7 CHECK\_EPOS Built-In Procedure

---

**Purpose:** Checks that the specified position is valid and that no motion errors will be generated when moving to this position

**Syntax:** CHECK\_EPOS(eposn, uframe, utool, status <, group\_no>)

Input/Output Parameters:

[in] eposn : XYZWPREXT

[in] uframe : POSITION

[in] utool : POSITION

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- eposn is the XYZWPREXT position to be checked.
- uframe specifies the uframe position to use with eposn.
- utool specifies the utool position to use with eposn.
- status explains the status of the check. If the position is reachable, the status will be 0.
- group\_no is optional, but if specified will be the group number for eposn. If not specified the default group of the program is used.

See Also: [Section A.7.11, GET\\_POS\\_FRM Built-In Procedure](#)

## A.3.8 CHECK\_NAME Built-In Procedure

**Purpose:** Checks a specified file or program name for illegal characters.

**Syntax:** CHECK\_NAME (name\_spec, status)

Input/Output Parameters:

[in] name\_spec : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- Name\_spec specifies the string to check for illegal characters. The string can be the file name or program name. It should not include the extension of the file or the program. This built-in does not handle special system names such as \*SYSTEM\*.

## A.3.9 CHR Built-In Function

**Purpose:** Returns the character that corresponds to a numeric code

**Syntax:** CHR (code)

Function Return Type : STRING

Input/Output Parameters:

[in] code : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- code represents the numeric code of the character for either the ASCII, Graphic, or Multinational character set.
- Returns a single character string that is assigned the value of code.

See Also: [Appendix D, CHARACTER CODES](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.4, STANDARD ROUTINES \(ROUT\\_EX.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.3.10 CLEAR Built-In Procedure

---

**Purpose:** Clears the specified program and/or variables from memory

**Syntax:** CLEAR(file\_spec, status)

Input/Output Parameters:

[in] file\_spec : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- file\_spec specifies the program name and type of data to clear. The following types are valid:
  - no ext : KAREL or Teach Pendant program and variables
  - .TP : Teach Pendant program
  - .PC : KAREL program
  - .VR : KAREL variables
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following example clears a KAREL program, clears the variables for a program, and clears a teach pendant program.

```
-- Clear KAREL program
CLEAR('test1.pc', status)
-- Clear KAREL variables
CLEAR('testvars.vr', status)
-- Clear Teach Pendant program
CLEAR('prg1.tp', status)
```

**Figure A.3.10 CLEAR Built-In Procedure**

## A.3.11 CLEAR\_SEMA Built-In Procedure

---

**Purpose:** Clear the indicated semaphore by setting the count to zero

**Syntax:** CLEAR\_SEMA(semaphore\_no)

Input/Output Parameters:

[in] semaphore\_no : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- The semaphore indicated by semaphore\_no is cleared.
- semaphore\_no must be in the range of 1 to the number of semaphores defined on the controller.
- All semaphores are cleared at COLD start. It is good practice to clear a semaphore prior to using it. Before several tasks begin sharing a semaphore, one and only one of these task, should clear the semaphore.

**See Also:** [Section A.16.18, POST\\_SEMA Built-In Procedure](#) , [Section A.16.7, PEND\\_SEMA Built-In Procedure](#) , [Section A.19.5, SEMA\\_COUNT Built-In Function](#) , examples in [Chapter 17, MULTI-TASKING](#)

## A.3.12 CLOSE FILE Statement

**Purpose:** Breaks the association between a FILE variable and a data file or communication port

**Syntax:** CLOSE FILE file\_var

where:

file\_var : a FILE variable

**Details:**

- file\_var must be a static variable that was used in the OPEN FILE statement.
- Any buffered data associated with the file\_var is written to the file or port.
- The built-in function IO\_STATUS will always return zero.

**See Also:** [Section A.9.19, IO\\_STATUS Built-In Function](#) , [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#) , [Appendix E, SYNTAX DIAGRAMS](#) , for additional syntax information

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.3.13 CLOSE HAND Statement

**Purpose:** Causes the specified hand to close

**Syntax:** CLOSE HAND hand\_num

where:

hand\_num :an INTEGER expression

**Details:**

- The actual effect of the statement depends on how the HAND signals are set up in I/O system.
- The valid range of values for hand\_num is 1-2. Otherwise, the program is aborted with an error.
- The statement has no effect if the value of hand\_num is in range but the hand is not connected.
- The program is aborted with an error if the value of hand\_num is in range but the HAND signal represented by that value has not been assigned.

**See Also:** [Chapter 16, INPUT/OUTPUT SYSTEM](#) , for more information on hand signals, [Appendix E, SYNTAX DIAGRAMS](#) , for additional syntax information

**Example:** The following example moves the robot to the first position and closes the hand specified by hand\_num.

```
SET_POS_REG(1, p1, status)
move_pr - Call TP program to do move
CLOSE HAND hand_num
```

**Figure A.3.13 CLOSE HAND Statement**

## A.3.14 CLOSE\_TPE Built-In Procedure

---

**Purpose:** Closes the specified teach pendant program

**Syntax:** CLOSE\_TPE(open\_id, status)

Input/Output Parameters:

[in] open\_id : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id indicates the teach pendant program to close. All teach pendant programs that are opened must be closed before they can be executed. Any unclosed programs remain opened until the KAREL program which opened it is aborted or runs to completion.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**See Also:** [Section A.15.3, OPEN\\_TPE Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.3.15 CLR\_IO\_STAT Built-In Procedure

---

**Purpose:** Clear the results of the last operation on the file argument

**Syntax:** CLR\_IO\_STAT(file\_id)

Input/Output Parameters:

[in] file\_id : FILE

%ENVIRONMENT Group : PBCORE

**Details:**

- Causes the last operation result on file\_id, which is returned by IO\_STATUS, to be cleared to zero.

**See Also:** [Section A.9.19, IO\\_STATUS Built-In Function](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.3.16 CLR\_PORT\_SIM Built-In Procedure

**Purpose:** Sets the specified port to be unsimulated

**Syntax:** CLR\_PORT\_SIM(port\_type, port\_no, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port to unsimulate. Codes are defined in FR:KLIOTYPS.KL.
- port\_no specifies the port number to unsimulate.
- status is returned with zero if parameters are valid and the simulation of the specified port is cleared.

**See Also:** [Section A.7.9, GET\\_PORT\\_SIM Built-In Procedure](#), [Section A.19.24, SET\\_PORT\\_SIM Built-In Procedure](#)

## A.3.17 CLR\_POS\_REG Built-In Procedure

**Purpose:** Removes all data for the specified group in the specified position register

**Syntax:** CLR\_POS\_REG(register\_no, group\_no, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] group\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the register number whose data should be cleared.
- If group\_no is zero, data for all groups is cleared.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#), [Section A.7.12, GET\\_POS\\_REG Built-In Function](#)

**Example:** The following example clears the first 100 position registers.

**Figure A.3.17 CLR\_POS\_REG Built-In Procedure**

```
FOR register_no = 1 to 100 DO
    CLR_POS_REG(register_no, 0, status)
ENDFOR
```

## A.3.18 %CMOSVARS Translator Directive

---

**Purpose:** Specifies the default storage for KAREL variables is permanent memory

**Syntax:** %CMOSVARS

**Details:**

- If %CMOSVARS is specified in the program, then all static variables by default will be created in permanent memory.
- If %CMOSVARS is not specified, then all static variables by default will be created in temporary memory.
- If a program specifies %CMOSVARS, but not all static variables need to be created in permanent memory, the IN DRAM clause can be used on selected variables.

**See Also:** [Section A.9.2, IN Clause](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DCLST\\_EX.KL\)](#)

## A.3.19 %CMOS2SHADOW Translator Directive

---

**Purpose:** Instructs the translator to put all CMOS variables in Shadow memory

**Syntax:** %CMOS2SHADOW

## A.3.20 CNC\_DYN\_DISB Built-In Procedure

---

**Purpose:** Cancels the dynamic display based on the value of a BOOLEAN variable in a specified window.

**Syntax:** CNC\_DYN\_DISB (b\_var, window\_name, status)

Input/Output Parameters:

[in] b\_var :BOOLEAN

[in] window\_name :STRING

[out] status :INTEGER

%ENVIRONMENT Group :UIF

**Details:**

- b\_var is the BOOLEAN variable whose dynamic display is to be canceled.
- window\_name must be a previously defined window name. See [Section 7.9.1, USER Menu on the Teach Pendant](#) and [Section 7.9.2, USER Menu on the CRT/KB](#) for predefined window names.
- If there is more than one display active for this variable in this window, all the displays are canceled.
- status returns an error if there is no dynamic display active specifying this variable and window. If not equal to 0, then an error occurred.

**See Also:** [Section A.9.5,INI\\_DYN\\_DISB Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## **A.3.21 CNC\_DYN\_DISE Built-In Procedure**

**Purpose:** Cancels the dynamic display based on the value of an INTEGER variable in a specified window.

**Syntax:** `CNC_DYN_DISE (e_var, window_name, status)`

Input/Output Parameters:

[in] `e_var` : INTEGER

[in] `window_name` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- `e_var` is the integer variable whose dynamic display is to be canceled.
- Refer to the [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.9.6,INI\\_DYN\\_DISE Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## **A.3.22 CNC\_DYN\_DISI Built-In Procedure**

**Purpose:** Cancels the dynamic display of an INTEGER variable in a specified window.

**Syntax:** `CNC_DYN_DISI(int_var, window_name, status)`

Input/Output Parameters:

[in] `int_var` : INTEGER

[in] `window_name` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- `int_var` is the INTEGER variable whose dynamic display is to be canceled.

- Refer to the [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.9.7,INI\\_DYN\\_DISI Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## A.3.23 CNC\_DYN\_DISP Built-In Procedure

---

**Purpose:** Cancels the dynamic display based on the value of a port in a specified window.

**Syntax:** CNC\_DYN\_DISP(port\_type, port\_no, window\_name, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] window\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- port\_type and port\_no are INTEGER values specifying the port whose dynamic display is to be canceled.
- Refer to the [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.9.8,INI\\_DYN\\_DISP Built-In Procedure](#) for information on port\_type codes.

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## A.3.24 CNC\_DYN\_DISR Built-In Procedure

---

**Purpose:** Cancels the dynamic display of a REAL number variable in a specified window.

**Syntax:** CNC\_DYN\_DISR(real\_var, window\_name, status)

Input/Output Parameters:

[in] real\_var : REAL

[in] window\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- `real_var` is the REAL variable whose dynamic display is to be canceled.
- Refer to the [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.9.9,INI\\_DYN\\_DISR Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## **A.3.25 CNC\_DYN\_DISS Built-In Procedure**

**Purpose:** Cancels the dynamic display of a STRING variable in a specified window.

**Syntax:** `CNC_DYN_DISS(str_var, window_name, status)`

Input/Output Parameters:

[in] `str_var` : STRING

[in] `window_name` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- `str_var` is the STRING variable whose dynamic display is to be canceled.
- Refer to the [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.9.10,INI\\_DYN\\_DISS Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## **A.3.26 CNCL\_STP\_MTN Built-In Procedure**

**Purpose:** Cancels all stopped motions

**Syntax:** `CNCL_STP_MTN`

%ENVIRONMENT Group : MOTN

- All stopped motions will be canceled for all groups that the program controls.
- The statements following the motion statements will be executed.

- CNCL\_STP\_MTN will have no effect if no motions are currently stopped.
- Motion cannot be canceled for a different task.

**Example:** The following example will cancel all stopped motions for all groups that the program controls after an emergency stop has occurred.

```
ROUTINE e_stop_hndlr
BEGIN
  CNCL_STP_MTN
END e_stop_hndlr
CONDITION[100]:
  WHEN ERROR[estop] DO
    UNPAUSE
    ENABLE CONDITION[100]
    e_stop_hndlr
  END CONDITION
ENABLE CONDITION[100]
```

**Figure A.3.26 CNCL\_STP\_MTN Built-In Procedure**

## A.3.27 CNV\_CNF\_STRG Built-In Procedure

**Purpose:** Converts the specified CONFIG into a STRING using an optional group\_no

**Syntax:** CNV\_CNF\_STRG(source, target, status <,group\_no>)

Input/Output Parameters :

- [in] source : CONFIG
- [out] target : STRING
- [out] status : INTEGER
- [in] group\_no : INTEGER

%ENVIRONMENT Group : STRNG

### Details:

- target receives the STRING form of the configuration specified by source.
- target must be long enough to accept a valid configuration string for the robot arm attached to the controller. Otherwise, the program will be aborted with an error.

Using a length of 25 is generally adequate because the longest configuration string of any robot is 25 characters long.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The config mask from group\_no is used during conversion. If group\_no is omitted, then default group for the program is assumed.

If group\_no is specified, it must be in the range 1 to the total number of groups defined on the controller.

**See Also:** [Section A.3.33, CNV\\_STR\\_CONF Built-In Procedure](#)

**Example:** The following example converts the configuration from position posn into a STRING and puts it into config\_string using the configuration mask from group 2. The STRING is then displayed on the screen.

```
CNV_CNF_STRG(posn.pos_config, config_string, 2)
WRITE('Configuration of posn in group 2: ', config_string, cr)
```

**Figure A.3.27 CNV\_CNF\_STRG Built-In Procedure**

## A.3.28 CNV\_CONF\_STR Built-In Procedure

**Purpose:** Converts the specified CONFIG into a STRING

**Syntax:** CNV\_CONF\_STR(source, target)

Input/Output Parameters:

[in] source :CONFIG

[out] target :STRING

%ENVIRONMENT Group :STRNG

**Details:**

- target receives the STRING form of the configuration specified by source.
- target must be long enough to accept a valid configuration string for the robot arm attached to the controller. Otherwise, the program will be aborted with an error.

Using a length of 25 is generally adequate because the longest configuration string of any robot is 25 characters long.

**See Also:** [Section A.3.33, CNV\\_STR\\_CONF Built-In Procedure](#)

**Example:** The following example converts the configuration from position posn into a STRING and puts it into config\_string. The STRING is then displayed on the screen.

**Figure A.3.28 CNV\_CONF\_STR Built-In Procedure**

```
CNV_CONF_STR(posn.pos_config, config_string)
WRITE('Configuration of posn: ', config_string, cr)
```

## A.3.29 CNV\_INT\_STR Built-In Procedure

**Purpose:** Formats the specified INTEGER into a STRING

**Syntax:** CNV\_INT\_STR(source, length, base, target)

Input/Output Parameters:

[in] source : INTEGER expression

[in] length : INTEGER expression

[in] base : INTEGER expression

[out] target : STRING expression

%ENVIRONMENT Group : PBCORE

**Details:**

- source is the INTEGER to be formatted into a STRING.
- length specifies the minimum length of the target. The actual length of target may be greater if required to contain the contents of source and at least one leading blank.
- base indicates the number system in which the number is to be represented. base must be in the range 2-16 or 0 (zero) indicating base 10.
- If the values of length or base are invalid, target is returned uninitialized.
- If target is not declared long enough to contain source and at least one leading blank, it is returned with one blank and the rest of its declared length filled with \*.

**See Also:** [Section A.3.34, CNV\\_STR\\_INT Built-In Procedure](#)

**Example:** Refer to the following section for detailed program examples:

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

---

### A.3.30 CNV\_JPOS\_REL Built-In Procedure

**Purpose:** Allows a KAREL program to examine individual joint angles as REAL values

**Syntax:** CNV\_JPOS\_REL(joint\_pos, real\_array, status)

Input/Output Parameters:

[in] joint\_pos : JOINTPOS

[out] real\_array : ARRAY [num\_joints] OF REAL

[out] status : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- joint\_pos is one of the KAREL joint position data types: JOINTPOS, or JOINTPOS1 through JOINTPOS9.
- num\_joints can be smaller than the number of joints in the system. A value of nine can be used if the actual number of joints is unknown. Joint number one will be stored in real\_array element number one, etc. Excess array elements will be ignored.
- The measurement of the real\_array elements is in degrees.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.32, CNV\\_REL\\_JPOS Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

---

### A.3.31 CNV\_REAL\_STR Built-In Procedure

**Purpose:** Formats the specified REAL value into a STRING

**Syntax:** CNV\_REAL\_STR(source, length, num\_digits, target)

Input/Output Parameters:

[in] source : REAL expression  
[in] length : INTEGER expression  
[in] num\_digits : INTEGER expression  
[out] target : STRING  
%ENVIRONMENT Group : STRNG

**Details:**

- source is the REAL value to be formatted.
- length specifies the minimum length of the target. The actual length of target may be greater if required to contain the contents of source and at least one leading blank.
- num\_digits specifies the number of digits displayed to the right of the decimal point. If num\_digits is a negative number, source will be formatted in scientific notation (where the ABS(num\_digits) represents the number of digits to the right of the decimal point.) If num\_digits is 0, the decimal point is suppressed.
- If length or num\_digits are invalid, target is returned uninitialized.
- If the declared length of target is not large enough to contain source with one leading blank, target is returned with one leading blank and the rest of its declared length filled with \*'s (asterisks).

See Also: [Section A.3.35, CNV\\_STR\\_REAL Built-In Procedure](#)

**Example:** The following example converts the REAL number in cur\_volts into a STRING and puts it into volt\_string. The minimum length of cur\_volts is specified to be seven characters with two characters after the decimal point. The contents of volt\_string is then displayed on the screen.

```
cur_volts = AIN[2]
CNV_REAL_STR(cur_volts, 7, 2, volt_string)
WRITE('Voltage=';volt_string,CR)
```

**Figure A.3.31 CNV\_REAL\_STR Built-In Procedure**

## A.3.32 CNV\_REL\_JPOS Built-In Procedure

**Purpose:** Allows a KAREL program to manipulate individual angles of a joint position

**Syntax:** CNV\_REL\_JPOS(real\_array, joint\_pos, status)

Input/Output Parameters:

[in] real\_array : ARRAY [num\_joints] OF REAL

[out] joint\_pos : JOINTPOS

[out] status : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- real\_array must have a declared size, equal to or greater than, the number of joints in the system. A value of nine can be used for num\_joints, if the actual number of joints is unknown. Array element number one will be stored in joint number one, and so forth. Excess array elements will be ignored. If the array is not large enough the program will abort with an invalid argument error.

- If any of the elements of real\_array that correspond to a joint angle are uninitialized, the program will be paused with an uninitialized variable error.
- The measurement of the real\_array elements is degrees.
- joint\_pos is one of the KAREL joint position types: JOINTPOS, or JOINTPOS1 through JOINTPOS9.
- joint\_pos receives the joint position form of real\_array.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

### A.3.33 CNV\_STR\_CONF Built-In Procedure

**Purpose:** Converts the specified configuration string into a CONFIG data type

**Syntax:** CNV\_STR\_CONF(source, target, status)

Input/Output Parameters:

[in] source : STRING

[out] target : CONFIG

[out] status : INTEGER

%ENVIRONMENT Group : STRNG

**Details:**

- target receives the CONFIG form of the configuration string specified by source.
- source must be a valid configuration string for the robot arm attached to the controller.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.28, CNV\\_CONF\\_STR Built-In Procedure](#)

**Example:** The following example sets the configuration of position xyz\_pos to the configuration specified by config\_string and then moves the TCP to that position.

```
CNV_STR_CONF(config_string, xyz_pos.config_data, status)
SET_POS_REG(1, xyz_pos, status) — Put xyz_pos in PR[1]
move_to_pr — Call TP program to move to PR[1]
```

**Figure A.3.33 CNV\_STR\_CONF Built-In Procedure**

### A.3.34 CNV\_STR\_INT Built-In Procedure

**Purpose:** Converts the specified STRING into an INTEGER

**Syntax:** CNV\_STR\_INT(source, target)

Input/Output Parameters:

[in] source : STRING

[out] target : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- source is converted into an INTEGER and stored in target.
- If source does not contain a valid representation of an INTEGER, target is set uninitialized.

**See Also:** [Section A.3.29, CNV\\_INT\\_STR Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

## A.3.35 CNV\_STR\_REAL Built-In Procedure

**Purpose:** Converts the specified STRING into a REAL

**Syntax:** CNV\_STR\_REAL(source, target)

Input/Output Parameters:

[in] source : STRING

[out] target : REAL

%ENVIRONMENT Group : PBCORE

**Details:**

- Converts source to a REAL number and stores the result in target.
- If source is not a valid decimal representation of a REAL number, target will be set uninitialized. source may contain scientific notation of the form nn.nnEsnn where s is a + or - sign.

**See Also:** [Section A.3.31, CNV\\_REAL\\_STR Built-In Procedure](#)

**Example:** The following example converts the STRING str into a REAL and puts it into rate.

```
REPEAT
    WRITE('Enter rate:')
    READ(str)
    CNV_STR_REAL(str, rate)
    UNTIL NOT UNINIT(rate)
```

**Figure A.3.35 CNV\_STR\_REAL Built-In Procedure**

## A.3.36 CNV\_STR\_TIME Built-In Procedure

**Purpose:** Converts a string representation of time to an integer representation of time.

**Syntax:** CNV\_STR\_TIME(source, target)

Input/Output Parameters:

[in] source : STRING

[out] target : INTEGER

%ENVIRONMENT Group : TIM

#### Details:

- The size of the string parameter, source, is STRING[20].
- source must be entered using DD-MMM-YYY HH:MM:SS format. The seconds specifier, SS, is optional. A value of zero (0) is used if seconds is not specified. If source is invalid, target will be set to 0.
- target can be used with the SET\_TIME built-in procedure to reset the time on the system. If target is 0, the time on the system will not be changed.

See Also: [Section A.19.33, SET\\_TIME Built-In Procedure](#)

**Example:** The following example converts the STRING variable str\_time, input by the user in DD-MMM-YYY HH:MM:SS format, to the INTEGER representation of time int\_time using the CNV\_STR\_TIME procedure. SET\_TIME is then used to set the time within the KAREL system to the time specified by int\_time.

```
WRITE('Enter the new time : ')
READ(str_time)
CNV_STR_TIME(str_time,int_time)
SET_TIME(int_time)
```

**Figure A.3.36 CNV\_STR\_TIME Built-In Procedure**

## A.3.37 CNV\_TIME\_STR Built-In Procedure

**Purpose:** Converts an INTEGER representation of time to a STRING

**Syntax:** CNV\_TIME\_STR(source, target)

Input/Output Parameters:

[in] source : INTEGER

[out] target : STRING

%ENVIRONMENT Group : TIM

#### Details:

- The GET\_TIME built-in procedure is used to determine the INTEGER representation of time. CNV\_TIME\_STR is used to convert source to target, which will be displayed in DD-MMM-YYY HH:MM:SS format.

See Also: [Section A.7.21, GET\\_TIME Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

## A.3.38 %COMMENT Translator Directive

**Purpose:** Specifies a comment of up to 16 characters

**Syntax:** %COMMENT = 'ssssssssssssss'

where sssssssssssssss = space

**Details:**

- The comment can be up to 16 characters long.
- During load time, the comment will be stored as a program attribute and can be displayed on the teach pendant or CRT/KB.
- %COMMENT must be used after the PROGRAM statement, but before any CONST, TYPE, or VAR sections.

**See Also:** [Section A.19.9, SET\\_ATTR\\_PRG Built-In Procedure](#) and [Section A.7.1, GET\\_ATTR\\_PRG Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

### **A.3.39 COMPARE\_FILE Built-in Procedure**

**Purpose:** Compares the contents of one file with another file

**Syntax:** COMPARE\_FILE(filea, fileb, result\_file, ascii\_flag, diff\_count, status)

Input/Output Parameters:

[in] filea : STRING

[in] fileb : STRING

[out] result\_file : FILE

[in] ascii\_flag: BOOLEAN

[out] diff\_count : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- filea specifies the device, name, and type of the file to compare. If no device is specified, the default device is used. You must specify both a name and type.
- fileb specifies the device, name, and type of the file to compare. If no device is specified, the default device is used. You must specify both a name and type.

- result\_file this must be an open FILE variable open for write. The comparison result will be written to this FILE.
- ascii\_flag specifies that the file(s) should be compared based on ASCII (text) content.
- diff\_count how many ASCII (text) lines differed.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.46, COPY\\_FILE Built-In Procedure](#) , [Section A.18.10, RENAME\\_FILE Built-In Procedure](#) , [Section A.4.12, DELETE\\_FILE Built-In Procedure](#)

The following example illustrates how this built-in can be used to detect if any TP programs have changed since a reference copy was saved to MC:\ref. To try this out copy md:\*.ls to mc:\ref (or another file storage area of your choosing) and then run this program.

```
-----
PROGRAM compfiles
-----
VAR
status: integer
prog_name: string[128]
logfile: file
file_ary: array [1638] of string[MAX_PROG_NAM]
tempstore in CMOS: string[40]
index: integer
findex: integer
n_progs: integer
diff_count: integer

BEGIN
  prog_name = '*'
  index = 0
  n_progs = 1
  diff_count=1
  IF uninit(tempstore) THEN
    tempstore = 'MC:\ref'
  ENDIF

  OPEN FILE logfile ('RW', 'CMPFILES.LOG')
  PROG_LIST(prog_name, 2, 0, 1, file_ary, n_progs, index)
  FOR index = 1 TO n_progs DO

    -- COMPARE_FILE( filename1: string; (IN) First file name for
    --               comparison
    --     filename2: string; (IN) Second file name for comparison
    --     out_file: file;   (OUT) Open KAREL file for output
    --     ascii_flag: boolean; (IN) If TRUE the comparison is ASCII
    --     count: integer; (OUT) ASCII- Number of differences/
    --                       Binary first different byte
    --     status: integer) (OUT) Result such as cannot open file

    COMPARE_FILE('MD:' + file_ary[index]+'.LS',
                  tempstore + '\'+ file_ary[index]+'.LS', logfile,
                  TRUE, diff_count, status)
    write logfile ('Compared ' + file_ary[index], ' ', status, ' ',
                  diff_count, ' ', index, CR)
  ENDFOR
```

```
CLOSE FILE logfile
END compfiles
```

Figure A.3.39 Example of COMPARE\_FILE built-in Procedure

## A.3.40 CONDITION...ENDCONDITION Statement

**Purpose:** Defines a global condition handler

**Syntax:** CONDITION[cond\_hand\_no]: [with\_list]

```
WHEN cond_list DO action_list
```

```
{WHEN cond_list DO action_list}
```

```
ENDCONDITION
```

**Details:**

- cond\_hand\_no specifies the number associated with the condition handler and must be in the range of 1-1000. The program is aborted with an error if it is outside this range.
- If a condition handler with the specified number already exists, the old one is replaced with the new one.
- The optional [with\_list] can be used to specify condition handler qualifiers. See the WITH clause for more information.
- All of the conditions listed in a single WHEN clause must be satisfied simultaneously for the condition handler to be triggered.
- Multiple conditions must all be separated by the AND operator or the OR operator. Mixing of AND and OR is not allowed.
- The actions listed after DO are to be taken when the corresponding conditions of a WHEN clause are satisfied simultaneously.
- Multiple actions are separated by a comma or on a new line.
- Calls to function routines are not allowed in a CONDITION statement.
- The condition handler is initially disabled and is disabled again whenever it is triggered. Use the ENABLE statement or action, specifying the condition handler number, to enable it.
- Use the DISABLE statement or action to deactivate a condition handler.
- The condition handler remains defined and can subsequently be reactivated by the ENABLE statement or action.
- The PURGE statement can be used to delete the definition of a condition handler.
- Condition handlers are known only to the task which defines them. Two different tasks can use the same cond\_hand\_no even though they specify different conditions.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , [Appendix E, SYNTAX DIAGRAMS](#) , for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.3.41 CONFIG Data Type

---

**Purpose:** Defines a variable or structure field as a CONFIG data type

**Syntax:** CONFIG

**Details:**

- CONFIG defines a variable or structure field as a highly compact structure consisting of fields defining a robot configuration.
- CONFIG contains the following predefined fields:
  - CFG\_TURN\_NO1 :INTEGER
  - CFG\_TURN\_NO2 :INTEGER
  - CFG\_TURN\_NO3 :INTEGER
  - CFG\_FLIP :BOOLEAN
  - CFG\_LEFT :BOOLEAN
  - CFG\_UP :BOOLEAN
  - CFG\_FRONT :BOOLEAN
- Variables and fields of structures can be declared as CONFIG.
- Subfields of CONFIG data type can be accessed and set using the usual structure field notation.
- Variables and fields declared as CONFIG can be
  - Assigned to one another.
  - Passed as parameters.
  - Written to and read from unformatted files.
- Each subfield of a CONFIG variable or structure field can be passed as a parameter to a routine, but is always passed by value.
- A CONFIG field is part of every POSITION and XYZWPR variable and field.
- An attempt to assign a value to a CONFIG subfield that is too large for the field results in an abort error.

**Example:** The following example shows how subfields of the CONFIG structure can be accessed and set using the usual structure.field notation.

**Figure A.3.41 CONFIG Data Type**

```

VAR
  config_var1, config_var2: CONFIG
  pos_var: POSITION
  seam_path: PATH
  i: INTEGER
BEGIN
  config_var1 = pos_var.config_data
  config_var1 = config_var2
  config_var1.cfg_turn_no1 = 0
  IF pos_var.config_data.cfg_flip THEN...
  FOR i = 1 TO PATH_LEN(seam_path) DO
    seam_path[i].node_pos.config_data = config_var1
  ENDFOR

```

## A.3.42 CONNECT TIMER Statement

**Purpose:** Causes an INTEGER variable to start being updated as a millisecond clock

**Syntax:** CONNECT TIMER TO clock\_var

where:

clock\_var : a static, user-defined INTEGER variable

**Details:**

- clock\_var is presently incremented by the value of the system variable \$SCR.\$COND\_TIME every \$SCR.\$COND\_TIME milliseconds as long as the program is running or paused and continues until the program disconnects the timer, ends, or aborts. For example, if \$SCR.\$COND\_TIME=32 then clock\_var will be incremented by 32 every 32 milliseconds.
- You should initialize clock\_var before using the CONNECT TIMER statement to ensure a proper starting value.
- If the variable is uninitialized, it will remain so for a short period of time (up to 32 milliseconds) and then it will be set to a very large negative value (-2.0E31 + 32 milliseconds) and incremented from that value.
- The program can reset the clock\_var to any value while it is connected.
- A clock\_var initialized at zero wraps around from approximately two billion to approximately minus two billion after about 23 days.
- If clock\_var is a system variable or a local variable in a routine, the program cannot be translated.

### NOTE

If two CONNECT TIMER statements using the same variable, are executed in two different tasks, the timer will advance twice as fast. For example, the timer will be incremented by  $2 * \$SCR.\$COND\_TIME$  every \$SCR.\$COND\_TIME ms. However, this does not occur if two or more CONNECT TIMER statements using the same variable, are executed in the same task.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for additional syntax information, [Section A.4.19, DISCONNECT TIMER Statement](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.3.43 CONTINUE Action

**Purpose:** Continues execution of a paused task

**Syntax:** CONTINUE <PROGRAM[n]>

**Details:**

- The CONTINUE action will not resume stopped motions.
- If program execution is paused, the CONTINUE action will continue program execution.
- The CONTINUE action can be followed by the clause PROGRAM[n], where n is the task number to be continued. Use GET\_TSK\_INFO to get a task number for a specified task name.

- A task can be in an interrupt routine when CONTINUE is executed. However, you should be aware of the following circumstances because CONTINUE only affects the current interrupt level, and interrupt levels of a task might be independently paused or running.
  - If the interrupt routine and the task are both paused, CONTINUE will continue the interrupt routine but the task will remain paused.
  - If the interrupt routine is running and the task is paused, CONTINUE will appear to have no effect because it will try to continue the running interrupt routine.
  - If the interrupt routine is paused and the task is running, CONTINUE will continue the interrupt routine.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.3.44 CONTINUE Condition

---

**Purpose:** Condition that is satisfied when program execution is continued

**Syntax:** CONTINUE <PROGRAM[n]>

**Details:**

- The CONTINUE condition monitors program execution.

If program execution is paused, the CONTINUE action, issuing CONTINUE from the CRT/KB or a CYCLE START from the operator panel, will continue program execution and satisfy the CONTINUE condition.

- The CONTINUE condition can be followed by the clause PROGRAM[n], where n is the task number to be continued. Use GET\_TSK\_INFO to get the task number of a specified task name.

**Example:** In the following example, program execution is being monitored. When the program is continued, a digital output will be turned on.

**Figure A.3.44 CONTINUE Condition**

```
CONDITION[1] :
  WHEN CONTINUE DO DOUT[1] = ON
ENDCONDITION
```

## A.3.45 CONT\_TASK Built-In Procedure

---

**Purpose:** Continues the specified task

**Syntax:** CONT\_TASK(task\_name, status)

**Input/Output Parameters:**

[in] task\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- task\_name is the name of the task to be continued. If the task was not paused, an error is returned in status.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- A task can be in an interrupt routine when CONT\_TASK is executed. However, you should be aware of the following circumstances because CONT\_TASK only affects the current interrupt level, and interrupt levels of a task might be independently paused or running.
  - If the interrupt routine and the task are both paused, CONT\_TASK will continue the interrupt routine but the task will remain paused.
  - If the interrupt routine is running and the task is paused, CONT\_TASK will appear to have no effect because it will try to continue the running interrupt routine.
  - If the interrupt routine is paused and the task is running, CONT\_TASK will continue the interrupt routine.

**See Also:** [Section A.18.43, RUN\\_TASK Built-In Procedure](#) , [Section A.1.4, ABORT\\_TASK Built-In Procedure](#) , [Section A.16.6, PAUSE\\_TASK Built-In Procedure](#) , [Chapter 17, MULTI-TASKING](#)

**Example:** The following example prompts the user for the task name and continues the task execution. Refer to [Chapter 17, MULTI-TASKING](#) , for more examples.

```
PROGRAM cont_task_ex
%ENVIRONMENT MULTI
  VAR
    task_str: STRING[12]
    status: INTEGER
  BEGIN
    WRITE('Enter task name to continue:')
    READ(task_str)
    CONT_TASK(task_str, status)
  END cont_task_ex
```

**Figure A.3.45 CONT\_TASK Built-In Procedure**

## A.3.46 COPY\_FILE Built-In Procedure

**Purpose:** Copies the contents of one file to another with the overwrite option

**Syntax:** COPY\_FILE(from\_file, to\_file, overwrite\_sw, nowait\_sw, status)

Input/Output Parameters:

[in] from\_file : STRING  
 [in] to\_file : STRING  
 [in] overwrite\_sw : BOOLEAN  
 [in] nowait\_sw : BOOLEAN  
 [out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- from\_file specifies the device, name, and type of the file from which to copy. from\_file can be specified using the wildcard (\*) character. If no device is specified, the default device is used. You must specify both a name and type. However, these can be a wildcard (\*) character.

- `to_file` specifies the device, name, and type of the file to which to copy. `to_file` can be specified using the wildcard (\*) character. If no device is specified, the default device is used.
- `overwrite_sw` specifies that the file(s) should be overwritten if they exist.
- If `nowait_sw` is TRUE, execution of the program continues while the command is executing. If it is FALSE, the program stops, including condition handlers, until the operation is complete. If you have time critical condition handlers in the program, put them in another program that executes as a separate task.
- If the program is aborted during the copy, the copy will completed before aborting.
- If the device you are copying to becomes full during the copy, an error will be returned.

**NOTE**

`nowait_sw` is not available in this release and should be set to FALSE.

- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.18.10, RENAME\\_FILE Built-In Procedure](#) , [Section A.4.12, DELETE\\_FILE Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

## A.3.47 COPY\_PATH Built-In Procedure

**Purpose:** Copies a complete path, part of a path, or a path in reverse node order (including associated data), to another identical type path variable.

**Syntax:** `COPY_PATH (source_path, start_node, end_node, dest_path, status)`

Input/Output Parameters:

[in] `source_path` : PATH

[in] `start_node` : INTEGER

[in] `end_node` : INTEGER

[in] `dest_path` : PATH

[out] `status` : INTEGER

%ENVIRONMENT Group : PATHOP

**Details:**

- `source_path` specifies the source path to copy from. This path can be a standard path or a user defined path.
- `start_node` specifies the number of the first node to copy. A value of 0 will copy the complete path, including header information. The `start_node` number must be between 0 and the highest node number in the source path. Otherwise, error status will be returned.
- `end_node` specifies the number of the last node to copy. A value of 0 will copy the complete path, including header information. The `end_node` number must be between 0 and the highest node number of the source path. Otherwise, error status will be returned.
- `dest_path` specifies the destination path to copy to. This path can be a standard path or a user defined path. However, the `dest_path` type must be identical to the `source_path` type. If they are not identical, an error status will be returned.

- status of 0 is returned if the parameters are valid and the COPY\_PATH operation was successful. Non-zero status indicates the COPY\_PATH operation was unsuccessful.

**NOTE**

To copy a complete path from one path variable to another identical path variable, set the start\_node and end\_node parameters to 0 (zero).

**An example of a partial path copy to a destination path.**

Executing the COPY\_PATH(P1, 2, 5, P2) command will copy node 2 through node 5 (inclusive) of path P1 to node 1 through 4 of Path P2, provided the path length of P1 is greater than or equal to 5. The destination path P2 will become a 4 node path. The original destination path is completely overwritten.

**An example of a source path copy in reverse order to a destination path.**

Executing the COPY\_PATH(P1, 5, 2, P2) command will copy node 5 through node 2 (inclusive) of path P1 to node 1 through 4 of Path P2, provided the path length of P1 is greater than or equal to 5. The destination path P2 will become a 4 node path. The original destination path is completely overwritten.

Specifically, the above command will copy node 5 of P1 to node 1 of P2, node 4 of P1 to node 2 of P2, and so forth.

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

### **A.3.48 COPY\_QUEUE Built-In Procedure**

**Purpose:** Copies one or more consecutive entries from a queue into an array of integers. The entries are not removed but are copied, starting with the oldest and proceeding to the newest, or until the output array, or integers, are full. A parameter specifies the number of entries at the head of the list (oldest entries) to be skipped.

**Syntax:** COPY\_QUEUE(queue, queue\_data, sequence\_no, n\_skip, out\_data, n\_got, status)

Input/Output Parameters:

[in] queue\_t : QUEUE\_TYPE

[in] queue\_data : ARRAY OF INTEGER

[in] n\_skip : INTEGER

[in] sequence\_no : INTEGER

[out] out\_data : ARRAY OF INTEGER

[out] n\_got : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBQMGR

**Details:**

- queue\_t specifies the queue variable for the queue from which the values are to be read.
- queue\_data specifies the array variable for the queue from which the values are to be read.

- sequence\_no specifies the sequence number of the oldest entry to be copied. If the sequence\_no is zero, the starting point for the copy is determined by the n\_skip parameter.
- n\_skip specifies the number of oldest entries to be skipped. A value of zero indicates to return the oldest entries.
- out\_data is an INTEGER array into which the values are to be copied; the size of the array is the maximum number of values returned.
- n\_got is returned with the number of entries returned. This will be one of the following:
  - Zero if there are n\_skip or fewer entries in the queue.
  - (queue\_to n\_entries\_skip) if this is less than ARRAY\_LEN(out\_data)
  - ARRAY\_LEN(out\_data) if this is less than or equal to queue.n\_entries - n\_skip
- status is returned with zero

**See Also:** [Section A.1.16, APPEND\\_QUEUE Built-In Procedure](#), [Section A.4.14, DELETE\\_QUEUE Built-In Procedure](#), [Section A.9.15, INSERT\\_QUEUE Built-In Procedure](#), [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

**Example:** The following example gets one page of a job queue and calls a routine, disp\_queue, to display this. If there are no entries for the page, the routine returns FALSE; otherwise the routine returns TRUE.

```

PROGRAM copy_queue_x
%environment PBQMGR
VAR
    job_queue FROM global_vars: QUEUE_TYPE
    job_data FROM global_vars: ARRAY[100] OF INTEGER
ROUTINE disp_queue(data: ARRAY OF INTEGER;
                    n_disp: INTEGER) FROM disp_prog
ROUTINE disp_page(data_array: ARRAY OF INTEGER;
                  page_no: INTEGER): BOOLEAN
VAR
    status: INTEGER
    n_got: INTEGER
BEGIN
    COPY_QUEUE(job_queue, job_data,
               (page_no - 1) * ARRAY_LEN(data_array), 0,
               data_array, n_got, status)
    IF (n_got = 0) THEN
        RETURN (FALSE)
    ELSE
        disp_queue(data_array, n_got)
        RETURN (TRUE)
    ENDIF
END disp_page
BEGIN
END copy_queue_x

```

**Figure A.3.48 COPY\_QUEUE Built-In Procedure**

## A.3.49 COPY\_TPE Built-In Procedure

**Purpose:** Copies one teach pendant program to another teach pendant program.

**Syntax:** COPY\_TPE(from\_prog, to\_prog, overwrite\_sw, status)

Input/Output Parameters:

[in] from\_prog : STRING  
[in] to\_prog : STRING  
[in] overwrite\_sw : BOOLEAN  
[out] status : INTEGER  
%ENVIRONMENT Group : TPE

**Details:**

- from\_prog specifies the teach pendant program name, without the .tp extension, to be copied.
- to\_prog specifies the new teach pendant program name, without the .tp extension, that from\_prog will be copied to.
- overwrite\_sw, if set to TRUE, will automatically overwrite the to\_prog if it already exists and it is not currently selected. If set to FALSE, the to\_prog will not be overwritten if it already exists.
- status explains the status of the attempted operation. If not equal to 0, the copy did not occur.

**See Also:** [Section A.3.52, CREATE\\_TPE Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.3.50 COS Built-In Function

**Purpose:** Returns the REAL cosine of the REAL angle argument, specified in degrees

**Syntax:** COS(angle)

Function Return Type: REAL

Input/Output Parameters:

[in] angle : REAL expression

%ENVIRONMENT Group : SYSTEM

**Details:**

- angle is an angle specified in the range of  $\pm 18000$  degrees. Otherwise, the program will be aborted with an error.

**Example:** Refer to the [Section A.20.1, TAN Built-In Function](#).

## A.3.51 CR Input/Output Item

**Purpose:** Can be used as a data item in a READ or WRITE statement to specify a carriage return

**Syntax:** CR

**Details:**

- When CR is used as a data item in a READ statement, it specifies that any remaining data in the current input line is to be ignored.

The next data item will be read from the start of the next input line.

- When CR is used as a data item in a WRITE statement, it specifies that subsequent output to the same file will appear on a new line.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.4, STANDARD ROUTINES \(ROUT\\_EX.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.3.52 CREATE\_TPE Built-In Procedure

---

**Purpose:** Creates a teach pendant program of the specified name

**Syntax:** CREATE\_TPE(prog\_name, prog\_type, status)

Input/Output Parameters:

[in] prog\_name : STRING

[in] prog\_type : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : TPE

### Details:

- prog\_name specifies the name of the program to be created.
- prog\_type specifies the type of the program to be created. The following constants are valid for program type:
  - PT\_MNE\_UNDEF : TPE program of undefined sub type
  - PT\_MNE\_JOB : TPE job
  - PT\_MNE\_PROC : TPE process
  - PT\_MNE\_MACRO : TPE macro
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred. Some of the possible errors are as follows:
  - 7015** Specified program exist
  - 9030** Program name is NULL

- **9031** Remove num from top of Program name
- **9032** Remove space from Program name
- **9036** Memory is not enough
- **9038** Invalid character in program name
- The program is created to reference all motion groups on the system. The program is created without any comment or any other program attributes. Once the program is created, SET\_ATTR\_PRG can be used to specify program attributes.

See Also: [Section A.19.9, SET\\_ATTR\\_PRG Built-In Procedure](#)

## A.3.53 CREATE\_VAR Built-In Procedure

**Purpose:** Creates the specified KAREL variable

**Syntax:** CREATE\_VAR(var\_prog\_nam, var\_nam, typ\_prog\_nam, type\_nam, group\_num, inner\_dim, mid\_dim, outer\_dim, status, <mem\_pool>)

Input/Output Parameters:

[in] var\_prog\_nam :STRING

[in] var\_nam :STRING

[in] typ\_prog\_nam :STRING

[in] type\_nam :STRING

[in] group\_num :INTEGER

[in] inner\_dim :INTEGER

[in] mid\_dim :INTEGER

[in] outer\_dim :INTEGER

[out] status :INTEGER

[in] mem\_pool :INTEGER

%ENVIRONMENT Group :MEMO

**Details:**

- var\_prog\_nam specifies the program name that the variable should be created in. If var\_prog\_nam is ' ', the default, which is the name of the program currently executing, is used.
- var\_nam specifies the variable name that will be created.
- If a variable is to be created as a user-defined type, the user-defined type must already be created in the system. typ\_prog\_nam specifies the program name of the user-defined type. If typ\_prog\_nam is ' ', the default, which is the name of the program currently executing, is used.
- type\_nam specifies the type name of the variable to be created. The following type names are valid:
  - 'ARRAY OF BYTE'
  - 'ARRAY OF SHORT'
  - 'BOOLEAN'
  - 'CAM\_SETUP'
  - 'CONFIG'
  - 'FILE'

- 'INTEGER'
- 'JOINTPOS'
- 'JOINTPOS1'
- 'JOINTPOS2'
- 'JOINTPOS3'
- 'JOINTPOS4'
- 'JOINTPOS5'
- 'JOINTPOS6'
- 'JOINTPOS7'
- 'JOINTPOS8'
- 'JOINTPOS9'
- 'MODEL'
- 'POSITION'
- 'REAL'
- 'STRING [n]', where n is the string length; the default is 12 if not specified.
- 'VECTOR'
- 'VIS\_PROCESS'
- 'XYZWPR'
- 'XYZWPREXT'
- Any other type names are considered user-defined types.
- group\_num specifies the group number to be used for positional data types.
- inner\_dim specifies the dimensions of the innermost array. For example, inner\_dim = 30 for ARRAY[10, 20, 30] OF INTEGER. inner\_dim should be set to 0 if the variable is not an array.
- mid\_dim specifies the dimensions of the middle array. For example, mid\_dim = 20 for ARRAY[10, 20, 30] OF INTEGER. mid\_dim should be set to 0 if the variable is not a 2-D array.
- outer\_dim specifies the dimensions of the outermost array. For example, outer\_dim = 10 for ARRAY[10, 20, 30] OF INTEGER. outer\_dim should be set to 0 if the variable is not a 3-D array.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- mem\_pool is an optional parameter that specifies the memory pool from which the variable is created. If not specified, then the variable is created in DRAM which is temporary memory. The DRAM variable must be recreated at every power up and the value is always reset to uninitialized.
- If mem\_pool = -1, then the variable is created in CMOS RAM which is permanent memory.

**See Also:** [Section A.3.10, CLEAR Built-In Procedure](#), [Section A.18.11, RENAME\\_VAR Built-In Procedure](#)

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.3.54 %CRTDEVICE Translator Directive

---

**Purpose:** Specifies that the CRT/KB device is the default device

**Syntax:** %CRTDEVICE

**Details:**

- Specifies that the INPUT/OUTPUT window will be the default in the READ and WRITE statements instead of the TPDISPLAY window.

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example showing how to use this device.

## A.3.55 CURJPOS Built-In Function

**Purpose:** Returns the current joint position of the tool center point (TCP) for the specified group of axes, even if one of the axes is in an overtravel

**Syntax:** CURJPOS(axs\_lim\_mask, ovr\_trv\_mask <,group\_no>)

Function Return Type : JOINTPOS

Input/Output Parameters:

[out] axs\_lim\_mask : INTEGER

[out] ovr\_trv\_mask : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- axs\_lim\_mask specifies which axes are outside the axis limits.
- ovr\_trv\_mask specifies which axes are in overtravel.

### NOTE

axis\_limit\_mask and ovr\_trv\_mask are not available in this release and can be set to 0.

**See Also:** [Section A.3.56, CURPOS Built-In Function](#) , [Chapter 8, POSITION DATA](#)

**Example:** The following example gets the current joint position of the robot.

```
PROGRAM getpos
VAR
  jnt: JOINTPOS
BEGIN
  jnt=CURJPOS(0,0)
END getpos
```

Figure A.3.55 CURJPOS Built-In Function

## A.3.56 CURPOS Built-In Function

**Purpose:** Returns the current Cartesian position of the tool center point (TCP) for the specified group of axes even if one of the axes is in an overtravel

**Syntax:** CURPOS(axis\_limit\_mask, ovr\_trv\_mask <,group\_no>)

Function Return Type : XYZWPREXT

Input/Output Parameters:

[out] axis\_limit\_mask :INTEGER

[out] ovr\_trv\_mask :INTEGER

[in] group\_no :INTEGER

%ENVIRONMENT Group :SYSTEM

**Details:**

- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- The group must be kinematic.
- Returns the current position of the tool center point (TCP) relative to the current value of the system variable \$UFRAME for the specified group.
- axis\_limit\_mask specifies which axes are outside the axis limits.
- ovr\_trv\_mask specifies which axes are in overtravel.

**NOTE**

axis\_limit\_mask and ovr\_trv\_mask are not available in this release and will be ignored if set.

**See Also:** [Chapter 8, POSITION DATA](#)

**Example:** Refer to the following program examples:

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.3.57 CURR\_PROG Built-In Function

**Purpose:** Returns the name of the program currently being executed

**Syntax:** CURR\_PROG

Function Return Type: STRING[12]

%ENVIRONMENT Group : BYNAM

**Details:**

- The variable assigned to CURR\_PROG must be declared with a string variable length  $\geq 12$

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.4 - D - KAREL LANGUAGE DESCRIPTION

## A.4.1 DAQ\_CHECKP Built-In Procedure

**Purpose:** To check the status of a pipe and the number of bytes available to be read from the pipe.

**Syntax:** DAQ\_CHECKP(pipe\_num, pipe\_stat, bytes\_avail)

Input/Output Parameters:

[in] pipe\_num : INTEGER

[out] pipe\_stat : INTEGER

[out] bytes\_avail : INTEGER

**Details:**

- pipe\_num is the number of the pipe (1 - 5) to check.
- pipe\_stat is the status of the pipe returned. The status is a combination of the following flags:
  - DAQ\_PIPREG is when the pipe is registered (value = 1).
  - DAQ\_ACTIVE is when the pipe is active (has been started) (value = 2).
  - DAQ\_CREATD is when the pipe is created (value = 4).
  - DAQ\_SNAPSH is when the pipe is in snapshot mode (value = 8).
  - DAQ\_1STRD is when the pipe has been read for the first time (value = 16).
  - DAQ\_OVFLOW is when the pipe is overflowed (value = 32).
  - DAQ\_FLUSH is when the pipe is being flushed (value = 64).
- bytes\_avail is the number of bytes that are available to be read from the pipe.
- The pipe\_stat returned parameter can be AND'ed with the above flag constants to determine whether the pipe is registered, is active, and so forth. For example, you must check to see if the pipe is active before writing to it.
- The DAQ\_OVFLOW flag will never be set for the task that writes to the pipe when it calls DAQ\_CHECKP. This flag applies only to tasks that read from the pipe.

**See Also:** [Section A.4.6, DAQ\\_WRITE Built-In Procedure](#)

**Example:** Refer to the DAQ\_WRITE example in the built-in function DAQ\_WRITE.

### NOTE

This built-in is available only when DAQ or data monitor options are loaded.

## A.4.2 DAQ\_REGPIPE Built-In Procedure

**Purpose:** To register a pipe for use in KAREL.

**Syntax :** DAQ\_REGPIPE(pipe\_num, mem\_type, pipe\_size, prog\_name, var\_name, pipe\_name, stream\_size, and status)

Input/Output Parameters :

[in] pipe\_num :INTEGER

[in] mem\_type :INTEGER

[in] pipe\_size :INTEGER

[in] prog\_name :STRING

[in] var\_name :STRING  
 [in] pipe\_name :STRING  
 [in] stream\_size :INTEGER  
 [out] status :INTEGER

**Details:**

- pipe\_num is the number of the pipe (1-5) to be registered.
- mem\_type allows you to allocate the memory to be used for the pipe. The following constants can be used:
  - DAQ\_DRAM allows you to allocate DRAM memory.
  - DAQ\_CMOS allows you to allocate CMOS memory.
- pipe\_size is the size of the pipe, is expressed as the number of data records that it can hold. The data record size itself is determined by the data type of var\_name.
- prog\_name is the name of the program containing the variable to be used for writing to the pipe. If passed as an empty string, the name of the current program is used.
- var\_name is the name of the variable that defines the data type to be used for writing to the pipe. Once registered, you can write any variable of this data type to the pipe.
- pipe\_name is the name of the pipe file. For example, if the pipe name is passed as 'foo.dat', the pipe will be accessible using the file string 'PIP:FOO.DAT'. A unique file name with an extension is required even if the pipe is being used only for sending to the PC.
- stream\_size is the number of records to automatically stream to an output file, if the pipe is started as a streamed pipe. A single write of the specified variable constitutes a single record in the pipe. If stream size is set to zero, the pipe will not automatically stream records to a file device; all data will be kept in the pipe until the pipe is read. Use stream\_size to help optimize network loading when the pipe is used to send data to the PC. If it is zero or one, the monitoring task will send each data record as soon as it is seen in the pipe. If the number is two or more, the monitor will wait until there are that many data records in the pipe before sending them all to the PC. In this manner, the overhead of sending network packets can be minimized. Data will not stay in the pipe longer than the time specified by the FlushTime argument supplied with the FRCPipe.StartMonitor method.
- status is the status of the attempted operation. If not 0, then an error occurred and the pipe was not registered.
- Pipes must be registered before they can be started and to which data is written. The registration operation tells the system how to configure the pipe when it is to be used. After it is registered, a pipe is configured to accept the writing of a certain amount of data per record, as governed by the size of the specified variable. In order to change the configuration of a pipe, the pipe must first be unregistered using DAQ\_UNREG, and then re-registered.

**See Also:** [Section A.4.5, DAQ\\_UNREG Built-In Procedure](#)

**Example:** The following example registers KAREL pipe 1 to write a variable in the program.

```
PROGRAM DAQREG
%ENVIRONMENT DAQ
CONST
  er_abort = 2
VAR
  status: INTEGER
  datavar: INTEGER
BEGIN
  -- Register pipe 1 DRAM as kldaq.dat
  -- It can hold 100 copies of the datavar variable
  -- before the pipe overflows
  DAQ_REGPIPE(1, DAQ_DRAM, 100, '', 'datavar', &
```

```
'kldaq.dat', 0, status)
IF status<>0 THEN
    POST_ERR(status, ' ', 0, er_abort)
ENDIF
END DAQREG
```

**Figure A.4.2 DAQ\_REGPIPE Built-In Procedure**

**NOTE**

This built-in is only available when DAQ or data monitor options are loaded.

## A.4.3 DAQ\_START Built-In Procedure

**Purpose:** To activate a KAREL pipe for writing.

**Syntax:** DAQ\_START(pipe\_num, pipe\_mode, stream\_dev, status)

Input/Output Parameters:

[in] pipe\_num : INTEGER  
[in] pipe\_mode : INTEGER  
[in] stream\_dev : STRING  
[out] status : INTEGER

**Details:**

- pipe\_num is the number of the pipe (1 - 5) to be started. The pipe must have been previously registered
- pipe\_mode is the output mode to be used for the pipe. The following constants are used:
  - DAQ\_SNAPSHT is the snapshot mode (each read of the pipe will result in all of the pipe's contents).
  - DAQ\_STREAM is the stream mode (each read from the same pipe file will result in data written since the previous read).
- stream\_dev is the device to which records will be automatically streamed. This parameter is ignored if the stream size was set to 0 during registration.
- status is the status of the attempted operation. If not 0, then an error occurred and the pipe was not unregistered.
- This built-in call can be made either from the same task/program as the writing task, or from a separate activate/deactivate task. The writing task can lie dormant until the pipe is started, at which point it begins to write data.
- A pipe is automatically started when a PC application issues the FRCPipe.StartMonitor method. In this case, there is no need for the KAREL application to call DAQ\_START to activate the pipe.
- Starting and stopping a pipe is tracked using a reference counting scheme. That is, any combination of two DAQ\_START and FRCPipe.StartMonitor calls requires any comb

**See Also:** [Section A.4.2, DAQ\\_REGPIPE Built-In Procedure](#) and [Section A.4.4, DAQ\\_STOP Built-In Procedure](#)

**Example:** The following example starts KAREL pipe 1 in streaming mode.

```

PROGRAM PIPONOFF
%ENVIRONMENT DAQ
CONST
    er_abort = 2
VAR
    status: INTEGER
    tpinput: STRING[1]
BEGIN
    -- prompt to turn on pipe
    WRITE('Press 1 to start pipe')
    READ (tpinput)
    IF tpinput = '1' THEN
        -- start pipe 1
        DAQ_START(1, DAQ_STREAM, 'RD:', status)
        IF status<>0 THEN
            POST_ERR(status, ' ', 0, er_abort)
        ELSE
            -- prompt to turn off pipe
            WRITE('Press any key to stop pipe')
            READ (tpinput)
            -- stop pipe 1
            DAQ_STOP(1, FALSE, status)
            IF status<>0 THEN
                POST_ERR(status, ' ', 0, er_abort)
            ENDIF
        ENDIF
    ENDIF
END PIPONOFF

```

**Figure A.4.3 DAQ\_START Built-In Procedure**

**NOTE**

This built-in is only available when DAQ or data monitor options are loaded.

## A.4.4 DAQ\_STOP Built-In Procedure

**Purpose:** To stop a KAREL pipe for writing.

**Syntax:** DAQ\_STOP(pipe\_num, force\_off, status)

Input/Output Parameters:

[in] pipe\_num : INTEGER

[in] force\_off : BOOLEAN

[out] status : INTEGER

**Details:**

- pipe\_num is the number of the pipe (1 - 5) to be stopped.
- force\_off occurs if TRUE force the pipe to be turned off, even if another application made a start request on the pipe. If set FALSE, if all start requests have been accounted for with stop requests, the pipe is turned off, else it remains on.

- status is the status of the attempted operation. If not 0, then an error occurred and the pipe was not stopped.
- The start/stop mechanism on each pipe works on a reference count. The pipe is started on the first start request, and each subsequent start request is counted. If a stop request is received for the pipe, the count is decremented.
- If the pipe is not forced off, and the count is not zero, the pipe stays on. By setting the force\_off flag to TRUE, the pipe is turned off regardless of the count. The count is reset.
- `FRCPipe.StopMonitor` method issued by a PC application is equivalent to a call to `DAQ_STOP`.

**See Also:** [Section A.4.3, DAQ\\_START Built-In Procedure](#)

**Example:** Refer to the `PIPONOFF` example in [Section A.4.3, DAQ\\_START Built-In Procedure](#).

**NOTE**

This built-in is only available when DAQ or data monitor options are loaded.

## A.4.5 DAQ\_UNREG Built-In Procedure

**Purpose:** To unregister a previously-registered KAREL pipe, so that it may be used for other data.

**Syntax:** `DAQ_UNREG(pipe_num, status)`

**Input/Output Parameters:**

[in] `pipe_num` : INTEGER

[out] `status` : INTEGER

**Details:**

- `pipe_num` is the number of the pipe (1 - 5) to be unregistered.
- `status` is the status of the attempted operation. If not 0, then an error occurred and the pipe was not unregistered.
- Unregistering a pipe allows the pipe to be re-configured for a different data size, pipe size, pipe name, and so forth. You must unregister the pipe before re-registering using `DAQ_REGPIPE`.

**See Also:** [Section A.4.2, DAQ\\_REGPIPE Built-In Procedure](#)

**Example:** The following example unregisters KAREL pipe 1.

```
PROGRAM DAQUNREG
%ENVIRONMENT DAQ
CONST
    er_abort = 2
VAR
    status:  INTEGER
BEGIN
    -- unregister pipe 1
    DAQ_UNREG(1, status)
    IF status<>0 THEN
        POST_ERR(status, ' ', 0, er_abort)
    ENDIF
END DAQUNREG
```

**Figure A.4.5 DAQ\_UNREG Built-In Procedure**

**NOTE**

This built-in is only available when DAQ or data monitor options are loaded.

## A.4.6 DAQ\_WRITE Built-In Procedure

**Purpose:** To write data to a KAREL pipe.

**Syntax:** DAQ\_WRITE(pipe\_num, prog\_name, var\_name, status)

Input/Output Parameters:

[in] pipe\_num : INTEGER  
 [in] prog\_name : STRING  
 [in] var\_name : STRING  
 [out] status : INTEGER

**Details:**

- pipe\_num is the number of the pipe (1 - 5) to which data is written.
- prog\_name is the name of the program containing the variable to be written. If passed as an empty string, the name of the current program is used.
- var\_name is the name of the variable to be written.
- status is the status of the attempted operation. If not 0, then an error occurred and the data was not written.
- You do not have to use the same variable for writing data to the pipe that was used to register the pipe. The only requirement is that the data type of the variable written matches the type of the variable used to register the pipe.
- If a PC application is monitoring the pipe, each call to DAQ\_WRITE will result in an FRCPipe\_Receive Event.

**See Also:** [Section A.4.2, DAQ\\_REGPIPE Built-In Procedure](#) and [Section A.4.1, DAQ\\_CHECKP Built-In Procedure](#)

**Example:** The following example registers KAREL pipe 2 and writes to it when the pipe is active.

```

PROGRAM DAQWRITE
%ENVIRONMENT DAQ
%ENVIRONMENT SYSDEF
CONST
  er_abort = 2
TYPE
  daq_data_t = STRUCTURE
    count: INTEGER
    dataval: INTEGER
  ENDSTRUCTURE
VAR
  status: INTEGER
  pipestat: INTEGER
  numbytes: INTEGER
  datavar: daq_data_t
BEGIN
  -- register 10KB pipe 2 in DRAM as kldaq.dat
  DAQ_REGPIPE(2, DAQ_DRAM, 100, '', 'datavar', &

```

```
'kldaq.dat', 1, status)
IF status<>0 THEN
    POST_ERR(status, ' ', 0, er_abort)
ENDIF
-- use DAQ_CHECKP to monitor status of pipe
DAQ_CHECKP(2, pipestat, numbytes)
datavar.count = 0
WHILE (pipestat AND DAQ_PIPREG) > 0 DO -- do while registered
    -- update data variable
    datavar.count = datavar.count + 1
    datavar.dataval = $FAST_CLOCK
    -- check if pipe is active
    IF (pipestat AND DAQ_ACTIVE) > 0 THEN
        -- write to pipe
        DAQ_WRITE(2, '', datavar, status)
        IF status<>0 THEN
            POST_ERR(status, ' ', 0, er_abort)
        ENDIF
    ENDIF
    -- put in delay to reduce loading
    DELAY(200)
    DAQ_CHECKP(2, pipestat, numbytes)
ENDWHILE
END DAQWRITE
```

**Figure A.4.6 DAQ\_WRITE Built-In Procedure**

**NOTE**

This built-in is only available when DAQ or data monitor options are loaded.

## A.4.7 %DEFGROUP Translator Directive

**Purpose:** Specifies the default motion group to be used by the translator

**Syntax:** %DEFGROUP = n

**Details:**

- n is the number of the motion group.
- The range is 1 to the number of groups on the controller.
- If %DEFGROUP is not specified, group 1 is used.

## A.4.8 DEF\_SCREEN Built-In Procedure

**Purpose:** Defines a screen

**Syntax:** DEF\_SCREEN(screen\_name, disp\_dev\_name, status)

Input/Output Parameters:

[in] screen\_name : STRING

[in] disp\_dev\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- Define a screen, associated with a specified display device, to which windows could be attached and be activated (displayed).
- screen\_name must be a unique, valid name (string), one to four characters long.
- disp\_dev\_name must be one of the display devices already defined, otherwise an error is returned. The following are the predefined display devices:
  - 'TP' – Teach Pendant Device
  - 'CRT' – CRT/KB Device
- status explains the status of the attempted operation. (If not equal to 0, then an error occurred.)

**See Also:** [Section A.1.7, ACT\\_SCREEN Built-In Procedure](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.4.9 DEF\_WINDOW Built-In Procedure

---

**Purpose:** Define a window

**Syntax:** DEF\_WINDOW(window\_name, n\_rows, n\_cols, options, status)

Input/Output Parameters:

[in] window\_name : STRING

[in] n\_rows : INTEGER

[in] n\_cols : INTEGER

[in] options : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Define a window that can be attached subsequently to a screen, have files opened to it, be written or have input echoed to it, and have information dynamically displayed in it.
- window\_name must be a valid name string, one to four characters long, and must not duplicate a window with the same name.
- n\_rows and n\_cols specify the size of the window in standard-sized characters. Any line containing double-wide or double-wide-double-high characters will contain only half this many characters. The first row and column begin at 1.
- options must be one of the following:
  - 0 : No option
  - wd\_com\_cursr : Common cursor
  - wd\_scrolled : Vertical scrolling
  - wd\_com\_cursr + wd\_scrolled : Common cursor + Vertical scrolling
- If common cursor is specified, wherever a write leaves the cursor is where the next write will go, regardless of the file variable used. Also, any display attributes set for any file variable associated with

this window will apply to all file variables associated with the window. If this is not specified, the cursor position and display attributes (except character size attributes, which always apply to the current line of a window) are maintained separately for each file variable open to the window. The common-cursor attribute is useful for windows that can be written to by more than one task and where these writes are to appear end-to-end. An example might be a log display.

- If vertical scrolling is specified and a line-feed, new-line, or index-down character is received and the cursor is in the bottom line of the window, all lines except the top line are moved up and the bottom line is cleared. If an index-up character is written, all lines except the bottom line are moved down and the top line is cleared. If this is not specified, the bottom or top line is cleared, but the rest of the window is unaffected.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.1.25, ATT\\_WINDOW\\_D Built-In Procedure](#) , [Section A.1.26, ATT\\_WINDOW\\_S Built-In Procedure](#)

## A.4.10 %DELAY Translator Directive

**Purpose:** Sets the amount of time program execution will be delayed every 250 milliseconds. Each program is delayed 8ms every 250ms by default. This allows the CPU to perform other functions such as servicing the CRT/KB and Teach Pendant user interfaces. %DELAY provides a way to change from the default and allow more CPU for system tasks such as user interface.

**Syntax:** %DELAY = n

**Details:**

- n is the delay time in milliseconds.
- The default delay time is 8 ms, if no DELAY is specified
- If n is set to 0, the program will attempt to use 100% of the available CPU time. This could result in the teach pendant and CRT/KB becoming inoperative since their priority is lower. A delay of 0 is acceptable if the program will be waiting for motion or I/O.
- While one program is being displayed, other programs are prohibited from executing. Interrupt routines (routines called from condition handlers) will also be delayed.
- Very large delay values will severely inhibit the running of all programs.
- To delay one program in favor of another, use the DELAY statement instead of %DELAY.

## A.4.11 DELAY Statement

**Purpose:** Causes execution of the program to be suspended for a specified number of milliseconds

**Syntax:** DELAY time\_in\_ms

where:

time\_in\_ms : an INTEGER expression

**Details:**

- If motion is active at the time of the delay, the motion continues.
- time\_in\_ms is the time in milliseconds. The actual delay will be from zero to \$SCR.\$cond\_time milliseconds less than the rounded time.
- A time specification of zero has no effect.

- If a program is paused while a delay is in progress, the delay will continue to be timed.
- If the delay time in a paused program expires while the program is still paused, the program, upon resuming and with no further delay, will continue execution with the statement following the delay. Otherwise, upon resumption, the program will finish the delay time before continuing execution.
- Aborting a program, or issuing RUN from the CRT/KB when a program is paused, terminates any delays in progress.
- While a program is awaiting expiration of a delay, the KCL> SHOW TASK command will show a hold of DELAY.
- A time value greater than one day or less than zero will cause the program to be aborted with an error.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES](#)  
(CHG\_DATA.KL)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING](#)  
(DOUT\_EX.KL)

## A.4.12 DELETE\_FILE Built-In Procedure

---

**Purpose:** Deletes the specified file

**Syntax:** DELETE\_FILE(file\_spec, nowait\_sw, status)

Input/Output Parameters:

[in] file\_spec : STRING

[in] nowait\_sw : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- file\_spec specifies the device, name, and type of the file to delete. file\_spec can be specified using the wildcard (\*) character. If no device name is specified, the default device is used.
- If nowait\_sw is TRUE, execution of the program continues while the command is executing. If it is FALSE, the program stops, including condition handlers, until the operation is complete. If you have time critical condition handlers in the program, put them in another program that executes as a separate task.

### NOTE

nowait\_sw is not available in this release and should be set to FALSE.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.46, COPY\\_FILE Built-In Procedure](#), [Section A.18.10, RENAME\\_FILE Built-In Procedure](#)

**Example:** Refer to [Section B.3, SAVING DATA TO THE DEFAULT DEVICE](#) (SAVE\_VRS.KL), for a detailed program example.

## A.4.13 DELETE\_NODE Built-In Procedure

---

**Purpose:** Deletes a path node from a PATH

**Syntax:** DELETE\_NODE(path\_var, node\_num, status)

Input/Output Parameters:

[in] path\_var : PATH

[in] node\_num : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PATHOP

**Details:**

- node\_num specifies the node to be deleted from the PATH specified by path\_var.
- All nodes past the deleted node will be renumbered.
- node\_num must be in the range from one to PATH\_LEN(path\_var). If it is outside this range, the status is returned with an error.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.1.15, APPEND\\_NODE Built-In Procedure](#), [Section A.9.14, INSERT\\_NODE Built-In Procedure](#)

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.4.14 DELETE\_QUEUE Built-In Procedure

---

**Purpose:** Deletes an entry from a queue

**Syntax:** DELETE\_QUEUE(sequence\_no, queue\_t, queue\_data, status)

Input/Output Parameters:

[in] sequence\_no : INTEGER

[in,out] queue\_t : QUEUE\_TYPE

[in,out] queue\_data : ARRAY OF INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBQMGR

**Details:**

- Use COPY\_QUEUE to get a list of the sequence numbers.
- sequence\_no specifies the sequence number of the entry to be deleted. Use COPY\_QUEUE to get a list of the sequence numbers.
- queue\_t specifies the queue variable for the queue.
- queue\_data specifies the array used to hold the data in the queue. The length of this array determines the maximum number of entries in the queue.
- status returns **61003**, Bad sequence no, if the specified sequence number is not in the queue.

**See Also:** [Section A.1.16, APPEND\\_QUEUE Built-In Procedure](#) , [Section A.3.48, COPY\\_QUEUE Built-In Procedure](#) , [Section A.9.15, INSERT\\_QUEUE Built-In Procedure](#) , [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

## A.4.15 DEL\_INST\_TPE Built-In Procedure

---

**Purpose:** Deletes the specified instruction in the specified teach pendant program

**Syntax:** `DEL_INST_TPE(open_id, lin_num, status)`

Input/Output Parameters:

[in] `open_id` : INTEGER

[in] `lin_num` : INTEGER

[out] `status` : INTEGER

%ENVIRONMENT Group : TPE

**Details:**

- `open_id` specifies the opened teach pendant program. A program must be opened with read/write access, using the `OPEN_TPE` built-in, before calling the `DEL_INST_TPE` built-in.
- `lin_num` specifies the line number of the instruction to be deleted.
- `status` explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**See Also:** [Section A.3.52, CREATE\\_TPE Built-In Procedure](#) , [Section A.3.14, CLOSE\\_TPE Built-In Procedure](#) , [Section A.3.49, COPY\\_TPE Built-In Procedure](#) , [Section A.15.3, OPEN\\_TPE Built-In Procedure](#) , [Section A.19.4, SELECT\\_TPE Built-In Procedure](#)

## A.4.16 DET\_WINDOW Built-In Procedure

---

**Purpose:** Detach a window from a screen

**Syntax:** `DET_WINDOW(window_name, screen_name, status)`

Input/Output Parameters:

[in] `window_name` : STRING

[in] `screen_name` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Removes the specified window from the specified screen.
- `window_name` and `screen_name` must be valid and already defined.
- The areas of other window(s) hidden by this window are redisplayed. Any area occupied by this window and not by any other window is cleared.
- An error occurs if the window is not attached to the screen.
- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.4.9, DEF\\_WINDOW Built-In Procedure](#) , [Section A.1.26, ATT\\_WINDOW\\_S Built-In Procedure](#) , [Section A.1.25, ATT\\_WINDOW\\_D Built-In Procedure](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.4.17 DISABLE CONDITION Action

**Purpose:** Used within a condition handler to disable the specified condition handler

**Syntax:** DISABLE CONDITION [cond\_hand\_no]

where:

cond\_hand\_no : an INTEGER expression

**Details:**

- If the condition handler is not defined, DISABLE CONDITION has no effect.
- If the condition handler is defined but not currently enabled, DISABLE CONDITION has no effect.
- When a condition handler is disabled, its conditions are not tested. Thus, if it is activated again, the conditions must be satisfied after the activation.
- Use the ENABLE CONDITION statement or action to reactivate a condition handler that has been disabled.
- cond\_hand\_no must be in the range of 1-1000. Otherwise, the program will be aborted with an error.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , for more information on using DISABLE CONDITION in condition handlers

**Example:** The following example disables condition handler number 2 when condition number 1 is triggered.

```
CONDITION[1] :  
  WHEN EVENT[1] DO  
    DISABLE CONDITION[2]  
  ENDCONDITION
```

Figure A.4.17 DISABLE CONDITION Action

## A.4.18 DISABLE CONDITION Statement

**Purpose:** Disables the specified condition handler

**Syntax:** DISABLE CONDITION [cond\_hand\_no]

where:

cond\_hand\_no : an INTEGER expression

**Details:**

- If the condition handler is not defined, DISABLE CONDITION has no effect.
- If the condition handler is defined but not currently enabled, DISABLE CONDITION has no effect.

- When a condition handler is disabled, its conditions are not tested. Thus, if it is activated again, the conditions must be satisfied after the activation.
- Use the `ENABLE CONDITION` statement or action to reactivate a condition handler that has been disabled.
- `cond_hand_no` must be in the range of 1-1000. Otherwise, the program will be aborted with an error.

**See Also:** [Chapter 6, CONDITION HANDLERS](#), for more information on using `DISABLE CONDITION` in condition handlers, [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information

**Example:** The following example allows the operator to choose whether or not to see count.

```
PROGRAM p_disable
VAR
    count      : INTEGER
    answer     : STRING[1]
ROUTINE showcount
BEGIN
    WRITE ('count = ',count::10,CR)
END showcount
BEGIN
    CONDITION[1]:
        WHEN EVENT[1] DO                  -- Condition[1] shows count
            showcount
            ENABLE CONDITION[1]
    ENDCONDITON
    ENABLE CONDITION[1]
    count = 0
    WRITE ('do you want to see count?')
    READ (answer,CR)
    IF answer = 'n'
        THEN DISABLE CONDITION[1]    -- Disables condition[1]
    ENDIF                         -- Count will not be shown
    FOR count = 1 TO 13 DO
        SIGNAL EVENT[1]
    ENDFOR
END p_disable
```

**Figure A.4.18 DISABLE CONDITION Statement**

## A.4.19 DISCONNECT TIMER Statement

**Purpose:** Stops updating a clock variable previously connected as a timer

**Syntax:** `DISCONNECT TIMER timer_var`

where:

`timer_var` : a static, user-defined INTEGER variable

**Details:**

- If `timer_var` is not currently connected as a timer, the `DISCONNECT TIMER` statement has no effect.
- If `timer_var` is a system or local variable, the program will not be translated.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information, `CONNECT TIMER Statement`

**Example:** The following example moves the TCP to the initial position in PR[1], sets the INTEGER variable timevar to 0 and connects the timer. After moving to the destination position in PR[2], the timer is disconnected.

```
move_to_pr1 - Call TP program to move to PR[1]
timevar = 0
CONNECT TIMER TO timevar

move_to_pr2 - Call TP program to move to PR[2]
DISCONNECT TIMER timevar
```

**Figure A.4.19 DISCONNECT TIMER Statement**

## A.4.20 DISCTRL\_ALPH Built-In Procedure

**Purpose:** Displays and controls alphanumeric string entry in a specified window.

**Syntax:** DISCTRL\_ALPH(window\_name, row, col, str, dict\_name, dict\_ele, term\_char, status)

Input/Output Parameters:

[in] window\_name : STRING

[in] row : INTEGER

[in] col : INTEGER

[in,out] str : STRING

[in] dict\_name : STRING

[in] dict\_ele : INTEGER

[out] term\_char : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : UIF

### Details:

- window\_name identifies the window where the str is currently displayed. See also [Section 7.9.1, USER Menu on the Teach Pendant](#) or [Section 7.9.2, USER Menu on the CRT/KB](#) for a listing of windows that may be used for window\_name.
- row specifies the row number where the str is displayed.
- col specifies the column number where the str is displayed.
- str specifies the KAREL string to be modified, which is currently displayed on the window\_name at position row and col.
- dict\_name specifies the dictionary that contains the words that can be entered. dict\_name can also be set to one of the following predefined values.
  - 'PROG' : program name entry
  - 'COMM' : comment entry
- dict\_ele specifies the dictionary element number for the words. dict\_ele can contain a maximum of 5 lines with no "&new\_line" accepted on the last line. See the example below.
- If a predefined value for dict\_name is used, then dict\_ele is ignored.

- term\_char receives a code indicating the character that terminated the menu. The code for key terminating conditions are defined in the include file FR:KLEVKEYS.KL. The following predefined constants are keys that are normally returned:
  - ky\_enter
  - ky\_prev
  - ky\_new\_menu
- DISCTRL\_ALPH will display and control string entry from the teach pendant device. To display and control string entry from the CRT/KB device, you must create an INTEGER variable, device\_stat, and set it to crt\_panel. To set control to the teach pendant device, set device\_stat to tp\_panel. Refer to the example below.
- status explains the status of an attempted operation. If not equal to 0, then an error occurred.

**NOTE**

DISCTRL\_ALPH will only display and control string entry if the **USER** or **USER2** menu is the selected menu. Therefore, use FORCE\_SPMENU(device\_stat, SPI\_TPUSER, 1) before calling DISCTRL\_ALPH to force the **USER** menu.

**See Also:** [Section A.1.7, ACT\\_SCREEN Built-In Procedure](#), [Section A.4.22, DISCTRL\\_LIST Built-In Procedure](#)

**Example:** Refer to [Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCLAP\\_EX.KL\)](#), for a detailed program example.

## A.4.21 DISCTRL\_FORM Built-In Procedure

**Purpose:** Displays and controls a form on the teach pendant or CRT/KB screen

**Syntax:** DISCTRL\_FORM(dict\_name, ele\_number, value\_array, inactive\_array, change\_array, term\_mask, def\_item, term\_char, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] ele\_number : INTEGER

[in] value\_array : ARRAY OF STRING

[in] inactive\_array : ARRAY OF BOOLEAN

[out] change\_array : ARRAY OF BOOLEAN

[in] term\_mask : INTEGER

[in,out] def\_item : INTEGER

[out] term\_char : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- dict\_name is the four-character name of the dictionary containing the form.
- ele\_number is the element number of the form.

- value\_array is an array of variable names that corresponds to each edit or display only data item in the form. Each variable name can be specified as a '`[prog_name] var_name`'.
  - [prog\_name] is the name of the program that contains the specified variable. If [prog\_name] is not specified, the current program being executed is used. '`[*SYSTEM*]`' should be used for system variables.
  - var\_name must refer to a static, global program variable.
  - var\_name can contain node numbers, field names, and/or subscripts.
  - var\_name can also specify a port variable with index. For example, '`DIN[1]`'.
- inactive\_array is an array of booleans that corresponds to each item in the form.
  - Each BOOLEAN defaults to FALSE, indicating it is active.
  - You can set any boolean to TRUE which will make that item inactive and non-selectable.
  - The array size can be greater than or less than the number of items in the form.
  - If an inactive\_array is not used, then an array size of 1 can be used. The array does not need to be initialized.
- change\_array is an array of booleans that corresponds to each edit or display only data item in the form.
  - If the corresponding value is set, then the boolean will be set to TRUE, otherwise it is set to FALSE. You do not need to initialize the array.
  - The array size can be greater than or less than the number of data items in the form.
  - If change\_array is not used, an array size of 1 can be used.
- term\_mask is a bit-wise mask indicating conditions that will terminate the form. This should be an OR of the constants defined in the include file `klevkmsk.kl`.
  - `kc_func_key` — Function keys
  - `kc_enter_key` — Enter and Return keys
  - `kc_prev_key` — PREV key

If either a selectable item or a new menu is selected, the form will always terminate, regardless of term\_mask.

- For version 6.20 and 6.21, def\_item receives the item you want to be highlighted when the form is entered. def\_item returns the item that was currently highlighted when the termination character was pressed.
- For version 6.22 and later, def\_item receives the item you want to be highlighted when the form is entered. def\_item is continuously updated while the form is displayed and contains the number of the item that is currently highlighted
- term\_char receives a code indicating the character or other condition that terminated the form. The codes for key terminating conditions are defined in the include file `klevkeys.kl`. Keys normally returned are pre-defined constants as follows:
  - `ky_undef` — No termination character was pressed
  - `ky_select` — A selectable item was selected
  - `ky_new_menu` — A new menu was selected
  - `ky_f1` — Function key 1 was selected
  - `ky_f2` — Function key 2 was selected
  - `ky_f3` — Function key 3 was selected
  - `ky_f4` — Function key 4 was selected
  - `ky_f5` — Function key 5 was selected
  - `ky_f6` — Function key 6 was selected
  - `ky_f7` — Function key 7 was selected

- ky\_f8 — Function key 8 was selected
- ky\_f9 — Function key 9 was selected
- ky\_f10 — Function key 10 was selected
- DISCTRL\_FORM will display the form on the teach pendant device. To display the form on the CRT/KB device, you must create an INTEGER variable, device\_stat, and set it to crt\_panel. To set control to the teach pendant device, set device\_stat to tp\_panel.
- status explains the status of the attempted operation. If status returns a value other than 0, an error has occurred.

**NOTE**

DISCTRL\_FORM will only display the form if the **USER2** menu is the selected menu. Therefore, use FORCE\_SPMENU(device\_stat, SPI\_TPUSER2, 1) before calling DISCTRL\_FORM to force the **USER2** menu.

**See Also:** [Chapter 11, DICTIONARIES AND FORMS](#), for more details and examples.

## A.4.22 DISCTRL\_LIST Built-In Procedure

**Purpose:** Displays and controls cursor movement and selection in a list in a specified window

**Syntax:** DISCTRL\_LIST(file\_var, display\_data, list\_data, action, status)

Input/Output Parameters:

[in] file\_var : FILE

[in,out] display\_data : DISP\_DAT\_T

[in] list\_data : ARRAY OF STRING

[in] action : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- file\_var must be opened to the window where the list data is to appear.
- display\_data is used to display the list. Refer to the DISP\_DAT\_T data type for details.
- list\_data contains the list of data to display.
- action must be one of the following:
  - dc\_disp : Positions cursor as defined in display\_data
  - dc\_up : Moves cursor up one row
  - dc\_dn : Moves cursor down one row
  - dc\_lf : Moves cursor left one field
  - dc\_rt : Moves cursor right one field
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- Using DISCTRL\_FORM is the preferred method for displaying and controlling information in a window.

**See Also:** [Section A.4.21, DISCTRL\\_FORM Built-In Procedure](#), [Section 7.9.1, USER Menu on the Teach Pendant](#), [Section 7.9.2, USER Menu on the CRT/KB](#), [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

**⚠ CAUTION**

The input parameters are not checked for validity. You must make sure the input parameters are valid; otherwise, the built-in might not work properly.

## A.4.23 DISCTRL\_PLMN Built-In Procedure

**Purpose:** Creates and controls cursor movement and selection in a pull-up menu

**Syntax:** DISCTRL\_PLMN(dict\_name, element\_no, ftn\_key\_no, def\_item, term\_char, status)

Input/Output Parameters:

[in] dict\_name : STRING  
[in] element\_no : INTEGER  
[in] ftn\_key\_num : INTEGER  
[in,out] def\_item : INTEGER  
[out] term\_char : INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- The menu data in the dictionary consists of a list of enumerated values that are displayed and selected from a pull-up menu on the teach pendant device. A maximum of 9 values should be used. Each value is a string of up to 12 characters.

A sequence of consecutive dictionary elements, starting with element\_no, define the values. Each value must be put in a separate element, and must not end with &new\_line. The characters are assigned the numeric values 1..9 in sequence. The last dictionary element must be "".

- dict\_name specifies the name of the dictionary that contains the menu data.
- element\_no is the element number of the first menu item within the dictionary.
- ftn\_key\_num is the function key where the pull-up menu should be displayed.
- def\_item is the item that should be highlighted when the menu is entered. 1 specifies the first item. On return, def\_item is the item that was currently highlighted when the termination character was pressed.
- term\_char receives a code indicating the character that terminated the menu. The codes for key terminating conditions are defined in the include file `FROM:KLEVKEYS.KL`. Keys normally returned are pre-defined constants as follows:
  - ky\_enter — A menu item was selected
  - ky\_prev — A menu item was not selected
  - ky\_new\_menu — A new menu was selected
  - ky\_f1

- ky\_f2
- ky\_f3
- ky\_f4
- ky\_f5
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** In this example, dictionary file TPEXAMEG.TX is loaded as 'EXAM' on the controller. TPPLMN.KL calls DISCTRL\_PLMN to display and process the pull-up menu above function key 3.

```
-----
TPEXAMEG.TX
-----
$subwin_menu
"Option 1"
$ 
"Option 2"
$ 
"Option 3"
$ 
"Option 4"
$ 
"Option 5"
$ 
"....."
-----
TPPLMN.KL
-----
PROGRAM tpplmn
%ENVIRONMENT uif
VAR
  def_item: INTEGER
  term_char: INTEGER
  status: INTEGER
BEGIN
  def_item = 1
  DISCTRL_PLMN('EXAM', 0, 3, def_item, term_char, status)
  IF term_char = ky_enter THEN
    WRITE (CR, def_item, ' was selected')
  ENDIF
END tpplmn
```

Figure A.4.23 DISCTRL\_PLMN Built-In Procedure

## A.4.24 DISCTRL\_SBMN Built-In Procedure

**Purpose:** Creates and controls cursor movement and selection in a sub-window menu

**Syntax:** DISCTRL\_SBMN(dict\_name, element\_no, def\_item, term\_char, status)

Input/Output Parameters:

[in] dict\_name : STRING  
 [in] element\_no : INTEGER  
 [in,out] def\_item : INTEGER

[out] term\_char : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : UIF

**Details:**

- The menu data in the dictionary consists of a list of enumerated values that are displayed and selected from the **sbm** subwindow on the Teach Pendant device. There can be up to 5 subwindow pages, for a maximum of 35 values. Each value is a string of up to 16 characters. If 4 or less enumerated values are used, then each string can be up to 40 characters.

A sequence of consecutive dictionary elements, starting with element\_no, define the values. Each value must be put in a separate element, and must not end with &new\_line. The characters are assigned the numeric values 1..35 in sequence. The last dictionary element must be "".

- dict\_name specifies the name of the dictionary that contains the menu data.
- element\_no is the element number of the first menu item within the dictionary.
- def\_item is the item that should be highlighted when the menu is entered. 1 specifies the first item. On return, def\_item is the item that was currently highlighted when the termination character was pressed.
- term\_char receives a code indicating the character that terminated the menu. The codes for key terminating conditions are defined in the include file FROM:KLEVKEYS.KL. Keys normally returned are pre-defined constants as follows:
  - ky\_enter — A menu item was selected
  - ky\_prev — A menu item was not selected
  - ky\_new\_menu — A new menu was selected
  - ky\_f1
  - ky\_f2
  - ky\_f3
  - ky\_f4
  - ky\_f5
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** In this example, dictionary file TPEXAMEG.TX is loaded as 'EXAM' on the controller. TPSBMN.KL calls DISCTRL\_SBMN to display and process the subwindow menu.

```
-----  
TPEXAMEG.TX  
-----  
$subwin_menu  
"Red"  
$  
"Blue"  
$  
"Green"  
$  
"Yellow"  
$  
"Brown"  
$  
"Pink"  
$  
"Mauve"  
$  
"Black"  
$
```

```

" Lime"
$
" Lemon"
$
" Beige"
$
" Blue"
$
" Green"
$
" Yellow"
$
" Brown"
$
"\a"
-----
TPSBMN.KL
-----
PROGRAM tpsbmn
%ENVIRONMENT uif
VAR
    def_item: INTEGER
    term_char: INTEGER
    status: INTEGER
BEGIN
    def_item = 1
    DISCTRL_SBMN('EXAM', 0, def_item, term_char, status)
    IF term_char = ky_enter THEN
        WRITE(CR, def_item, ' was selected')
    ENDIF
END tpsbmn

```

**Figure A.4.24 DISCTRL\_SBMN Built-In Procedure**

## A.4.25 DISCTRL\_TBL Built-In Procedure

**Purpose:** Displays and controls a table on the teach pendant

**Syntax:** DISCTRL\_TBL(dict\_name, ele\_number, num\_rows, num\_columns, col\_data, inact\_array, change\_array, def\_item, term\_char, term\_mask, value\_array, attach\_wind, status)

Input/Output Parameters:

- [in] dict\_name : STRING
- [in] ele\_number : INTEGER
- [in] num\_rows : INTEGER
- [in] num\_columns : INTEGER
- [in] col\_data : ARRAY OF COL\_DESC\_T
- [in] inact\_array : ARRAY OF BOOLEAN
- [out] change\_array : ARRAY OF BOOLEAN
- [in,out] def\_item : INTEGER
- [out] term\_char : INTEGER

[in] term\_mask : INTEGER  
[in] value\_array : ARRAY OF STRING  
[in] attach\_wind : BOOLEAN  
[out] status : INTEGER  
%ENVIRONMENT Group : UIF

**Details:**

- DISCTRL\_TBL is similar to the INIT\_TBL and ACT\_TBL built-In routines and should be used if no special processing needs to be done with each keystroke.
- dict\_name is the four-character name of the dictionary containing the table header.
- ele\_number is the element number of the table header.
- num\_rows is the number of rows in the table.
- num\_columns is the number of columns in the table.
- col\_data is an array of column descriptor structures, one for each column in the table. For a complete description, refer to the [Section A.9.12, INIT\\_TBL Built-In Procedure](#) in this appendix.
- inact\_array is an array of booleans that corresponds to each column in the table.
  - You can set each boolean to TRUE which will make that column inactive. This means the you cannot move the cursor to this column.
  - The array size can be less than or greater than the number of items in the table.
  - If inact\_array is not used, then an array size of 1 can be used, and the array does not need to be initialized.
- change\_array is a two-dimensional array of BOOLEANS that corresponds to formatted data items in the table.
  - If the corresponding value is set, then the boolean will be set to TRUE, otherwise it is set to FALSE. You do not need to initialize the array.
  - The array size can be less than or greater than the number of data items in the table.
  - If change\_array is not used, then an array size of 1 can be used.
- def\_item is the row containing the item you want to be highlighted when the table is entered. On return, def\_item is the row containing the item that was currently highlighted when the termination character was pressed.
- term\_char receives a code indicating the character or other condition that terminated the table. The codes for key terminating conditions are defined in the include file FROM: KLEVKEYS.KL. Keys normally returned are pre-defined constants as follows:
  - ky\_undef — No termination character was pressed
  - ky\_select — A selectable item as selected
  - ky\_new\_menu — A new menu was selected
  - ky\_f1 — Function key 1 was selected
  - ky\_f2 — Function key 2 was selected
  - ky\_f3 — Function key 3 was selected
  - ky\_f4 — Function key 4 was selected
  - ky\_f5 — Function key 5 was selected
  - ky\_f6 — Function key 6 was selected
  - ky\_f7 — Function key 7 was selected
  - ky\_f8 — Function key 8 was selected
  - ky\_f9 — Function key 9 was selected
  - ky\_f10 — Function key 10 was selected

- term\_mask is a bit-wise mask indicating conditions that will terminate the request. This should be an OR of the constants defined in the include file FROM:KLEVVKMSK.KL.
  - kc\_display — Displayable keys
  - kc\_func\_key — Function keys
  - kc\_keypad — Key-pad and Edit keys
  - kc\_enter\_key — Enter and Return keys
  - kc\_delete — Delete and Backspace keys
  - kc\_lr\_arw — Left and Right Arrow keys
  - kc\_ud\_arw — Up and Down Arrow keys
  - kc\_other — Other keys (such as Prev)
- value\_array is an array of variable names that corresponds to each column of data item in the table. Each variable name can be specified as '[prog\_name]var\_name'.
  - [prog\_name] specifies the name of the program that contains the specified variable. If [prog\_name] is not specified, then the current program being executed is used.
  - var\_name must refer to a static, global program variable.
  - var\_name can contain node numbers, field names, and/or subscripts.
- attach\_wind should be set to 1 if the table manager window needs to be attached to the display device. If it is already attached, this parameter can be set to 0.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Refer to the [Section A.9.12, INIT\\_TBL Built-In Procedure](#) for an example of setting up the dictionary text and initializing the parameters.

## A.4.26 DISMOUNT\_DEV Built-In Procedure

---

**Purpose:** Dismounts the specified device.

**Syntax:** DISMOUNT\_DEV (device, status)

Input/Output Parameters:

[in] device : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- device specifies the device to be dismounted.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.13.4, MOUNT\\_DEV Built-In Procedure](#), [Section A.6.6, FORMAT\\_DEV Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

## A.4.27 DISP\_DAT\_T Data Type

---

**Purpose:** Defines data type for use in DISCTRL\_LIST built-in

**Syntax:**

```
disp_dat_t = STRUCTURE
  win_start : ARRAY [4] OF SHORT
  win_end : ARRAY [4] OF SHORT
  curr_win : SHORT
  cursor_row : SHORT
  lins_per_pg : SHORT
  curs_st_col : ARRAY [10] OF SHORT
  curs_en_col : ARRAY [10] OF SHORT
  curr_field : SHORT
  last_field : SHORT
  curr_it_num : SHORT
  sob_it_num : SHORT
  eob_it_num : SHORT
  last_it_num : SHORT
  menu_id : SHORT
ENDSTRUCTURE
```

**Details:**

- disp\_dat\_t can be used to display a list in four different windows. The list can contain up to 10 fields. Left and right arrows move between fields. Up and down arrows move within a field.
- win\_start is the starting row for each window.
- win\_end is the ending row for each window.
- curr\_win defines the window to display. The count begins at zero (0 will display the first window).
- cursor\_row is the current cursor row.
- lins\_per\_pg is the number of lines per page for paging up and down.
- curs\_st\_col is the cursor starting column for each field. The range is 0-39 for the teach pendant.
- curs\_en\_col is the cursor ending column for each field. The range is 0-39 for the teach pendant.
- curr\_field is the current field in which the cursor is located. The count begins at zero (0 will set the cursor to the first field).
- last\_field is the last field in the list.
- curr\_it\_num is the item number the cursor is on.
- sob\_it\_num is the item number of the first item in the array.
- eob\_it\_num is the item number of the last item in the array.
- last\_it\_num is the item number of the last item in the list.
- menu\_id is the current menu identifier. Not implemented. May be left uninitialized.

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLIST\\_EX.KL\)](#), for a detailed program example.

## A.4.28 DOSFILE\_INF Built-In Procedure

---

**Purpose:** Returns information for a device as a string in the value parameter.

**Syntax:** DOSFILE\_INF(device, item, value\_str, status)

Input/Output Parameters:

[in] device : STRING

[in] item : INTEGER

[out] value\_str : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- DOSFILE\_INF is only available on R-30iB Controller
- device can be UD1 :
- item is the constant of the data to be returned:
  - DF\_MANUFNAME (Manufacturer's name)
  - DF\_PRODNAME (Product name)
  - DF\_SERIALNO (Serial number)
  - DF\_PRODID (Product ID number as assigned by USB authority)
  - DF\_VENDID (Vendor ID number as assigned by USB authority)
- value\_str is the string where the data is returned.
- status explains the status of an attempted operation. If not equal to 0, then an error occurred.

**Example:** The following example gets information about the UD1 device.

```
program dfinfo
%NOLOCKGROUP
var
value: string[60]
status: integer
begin
value = ''
DOSFILE_INF('UD1:', DF_MANUFNAME, value, status)
write (cr, '1: ', status, ', ', value, cr)
value = ''
DOSFILE_INF('UD1:', DF_PRODNAME, value, status)
write (cr, '2: ', status, ', ', value, cr)
value = ''
DOSFILE_INF('UD1:', DF_SERIALNO, value, status)
write (cr, '3: ', status, ', ', value, cr)
delay (2000)
value = ''
DOSFILE_INF('UD1:', DF_PRODID, value, status)
write (cr, '4: ', status, ', ', value, cr)
value = ''
DOSFILE_INF('UD1:', DF_VENDID, value, status)
```

```
write (cr, '5: ', status, ', ', value, cr)
end dfinfo
```

Figure A.4.28 DOSFILE\_INF Built-In Procedure

## A.5 - E - KAREL LANGUAGE DESCRIPTION

### A.5.1 ENABLE CONDITION Action

**Purpose:** Enables the specified condition handler

**Syntax:** ENABLE CONDITION [cond\_hand\_no]

where:

cond\_hand\_no : an INTEGER expression

**Details:**

- ENABLE CONDITION has no effect when
  - The condition handler is not defined
  - The condition handler is defined but is already enabled
- cond\_hand\_no must be in the range of 1-1000. Otherwise, the program will be aborted with an error.
- When a condition handler is enabled, its conditions are tested each time the condition handler is scanned. If the conditions are satisfied, the corresponding actions are performed and the condition handler is deactivated. Issue an ENABLE CONDITION statement or action to reactivate it.
- Use the DISABLE CONDITION statement or action to deactivate a condition handler that has been enabled.
- Condition handlers are known only to the task which defines them. One task cannot enable another tasks condition.

**See Also:** [Section A.4.17, DISABLE CONDITION Action , Chapter 6, CONDITION HANDLERS](#)

**Example:** Refer to ([Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

### A.5.2 ENABLE CONDITION Statement

**Purpose:** Enables the specified condition handler

**Syntax:** ENABLE CONDITION [cond\_hand\_no]

where:

cond\_hand\_no : an INTEGER expression

**Details:**

- ENABLE CONDITION has no effect when
  - The condition handler is not defined
  - The condition handler is defined but is already enabled

- cond\_hand\_no must be in the range of 1-1000. Otherwise, the program will be aborted with an error.
- When a condition handler is enabled, its conditions are tested each time the condition handler is scanned. If the conditions are satisfied, the corresponding actions are performed and the condition handler is deactivated. Issue an ENABLE CONDITION statement or action to reactivate it.
- Use the DISABLE CONDITION statement or action to deactivate a condition handler that has been enabled.
- Condition handlers are known only to the task which defines them. One task cannot enable another tasks condition.

**See Also:** [Section A.4.18, DISABLE CONDITION Statement](#) , [Chapter 6, CONDITION HANDLERS](#) , [Appendix E, SYNTAX DIAGRAMS](#) , for additional syntax information

**Example:** Refer to the following sections for detailed program examples.

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOV.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.5.3 %ENVIRONMENT Translator Directive

---

**Purpose:** Loads environment file.

**Syntax:** %ENVIRONMENT path\_name

- Used by the off-line translator to specify that the binary file, path\_name.ev, should be loaded. Environment files contain definitions for predefined constants, ports, types, system variables, and built-ins.
- All .EV files are loaded upon installation of the controller software. Therefore, the controller's translator will ignore %ENVIRONMENT statements since it already has the .EV files loaded.
- path\_name can be one of the following:
  - BYNAM
  - CTDEF (allows program access to CRT/KB system variables)
  - ERRS
  - FDEV
  - FLBT
  - IOSETUP
  - KCLOP
  - MEMO
  - MIR
  - MOTN
  - MULTI
  - PATHOP
  - PBCORE
  - PBQMGR
  - REGOPE
  - STRNG
  - SYSDEF (allows program access to most system variables)
  - SYSTEM

- TIM
  - TPE
  - TRANS
  - UIF
  - VECTR
- If no %ENVIRONMENT statements are specified in your KAREL program, the off-line translator will load all the .EV files specified in TRMNEG.TX. The translator must be able to find these files in the current directory or in one of the PATH directories.
  - If at least one %ENVIRONMENT statement is specified, the off-line translator will only load the files you specify in your KAREL program. Specifying your own %ENVIRONMENT statements will reduce the amount of memory required to translate and will be faster, especially if you do not require system variables since SYSDEF.EV is the largest file.
  - SYSTEM.EV and PBCORE.EV are automatically loaded by the translator and should not be specified in your KAREL program. The off-line translator will print the message Continuing without system defined symbols if it cannot find SYSTEM.EV. Do not ignore this message. Make sure the SYSTEM.EV file is loaded.

**Example:** Refer to the following sections for detailed program examples:

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.5.4 ERR\_DATA Built-In Procedure

**Purpose:** Reads the requested error from the error history and returns the error

**Syntax:** ERR\_DATA(seq\_num, error\_code, error\_string, cause\_code, cause\_string, time\_int, severity, prog\_nam)

Input/Output Parameters:

[in,out] seq\_num : INTEGER

[out] error\_code : INTEGER

[out] error\_string : STRING

[out] cause\_code : INTEGER

[out] cause\_string : STRING

[out] time\_int : INTEGER

[out] severity : INTEGER

[out] prog\_nam : STRING

%ENVIRONMENT Group : ERRS

- seq\_num is the sequence number of the previous error requested. seq\_num should be set to 0 if the oldest error in the history is desired. seq\_num should be set to MAXINT if the most recent error is desired.
- seq\_num is set to the sequence number of the error that is returned.

- If the initial value of seq\_num is greater than the sequence number of the newest error in the log, seq\_num is returned as zero and no other data is returned.
- If the initial value of seq\_num is less than the sequence number of the oldest error in the log, the oldest error is returned.
- error\_code returns the error code and error\_string returns the error message. error\_string must be at least 40 characters long or the program will abort with an error.
- cause\_code returns the reason code if it exists and cause\_string returns the message. cause\_string must be at least 40 characters long or the program will abort with an error.
- error\_code and cause\_code are in the following format:

ffccc (decimal)

where ff represents the facility code of the error.

ccc represents the error code within the specified facility.

Refer to [Chapter 6, CONDITION HANDLERS](#), for the error facility codes.

- time\_int returns the time that error\_code was posted. The time is in encoded format, and CNV\_TIME\_STR Built-In should be used to get the date-and-time string.
- severity returns one of the following error\_codes:
  - **0** : WARNING
  - **1** : PAUSE
  - **2** : ABORT
- If the error occurs in the execution of a program, prog\_nam specifies the name of the program in which the error occurred.
- If the error is posted by POST\_ERR, or if the error is not associated with a particular program (E-STOP, for example), prog\_nam is returned as "".
- Calling ERR\_DATA immediately after POST\_ERR may not return the error just posted since POST\_ERR returns before the error is actually in the error log.

**See Also:** [Section A.16.16, POST\\_ERR Built-In Procedure](#)

## A.5.5 ERROR Condition

**Purpose:** Specifies an error as a condition

**Syntax:** ERROR[n]

where:

n : an INTEGER expression or asterisk (\*)

**Details:**

- If n is an INTEGER, it represents an error code number. The condition is satisfied when the specified error occurs.
- If n is an asterisk (\*), it represents a wildcard. The condition is satisfied when any error occurs.
- The condition is an event condition, meaning it is satisfied only for the scan performed when the error was detected. The error is not remembered on subsequent scans.

**See Also:** [Chapter 6, CONDITION HANDLERS](#), for more information on using conditions. The *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)* for a list of all error codes

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.5.6 EVAL Clause

**Purpose:** Allows expressions to be evaluated in a condition handler definition

**Syntax:** EVAL(expression)

where:

expression : a valid KAREL expression

**Details:**

- expression is evaluated when the condition handler is defined, rather than dynamically during scanning.
- expression can be any valid expression that does not contain a function call.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , for more information on using conditions

**Example:** The following example causes delay until AIN[force] is greater than the evaluated expression (10 \* f\_scale).

```
WRITE ('Enter force scale: ')
READ (f_scale)
REPEAT
    DELAY(1000)
    UNTIL AIN[force] > EVAL(10 * f_scale)
```

**Figure A.5.6 EVAL Clause**

## A.5.7 EVENT Condition

**Purpose:** Specifies the number of an event that satisfies a condition when a SIGNAL EVENT statement or action with that event number is executed

**Syntax:** EVENT[event\_no]

where:

event\_no : is an INTEGER expression

**Details:**

- Events can be used as user-defined event codes that become TRUE when signaled.
- The SIGNAL EVENT statement or action is used to signal that an event has occurred.
- event\_no must be in the range of -32768 to 32767.

**See Also:** [Section A.19.41, SIGNAL EVENT Action](#) , [Section A.3.40, CONDITION...ENDCONDITION Statement](#) , [Section A.19.42, SIGNAL EVENT Statement](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.5.8 EXP Built-In Function

---

**Purpose:** Returns a REAL value equal to e (approximately 2.71828) raised to the power specified by a REAL argument

**Syntax:** EXP(x)

Function Return Type : REAL

Input/Output Parameters :

[in] x : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- EXP returns e (base of the natural logarithm) raised to the power x.
- x must be less than 80. Otherwise, the program will be paused with an error.

**Example:** The following example uses the EXP built-in to evaluate the exponent of the expression  $(-6.44 + \text{timevar}/(\text{timevar} + 20))$ .

```
WRITE (CR, 'Enter time needed for move:')
READ (timevar)
distance = timevar *
           EXP(-6.44 + timevar/(timevar + 20))
WRITE (CR, CR, 'Distance for move:', distance::10::3)
```

Figure A.5.8 EXP Built-In Function

## A.6 - F - KAREL LANGUAGE DESCRIPTION

---

### A.6.1 FILE Data Type

---

**Purpose:** Defines a variable as FILE data type

**Syntax:** FILE

**Details:**

- FILE allows you to declare a static variable as a file.
- You must use a FILE variable in OPEN FILE, READ, WRITE, CANCEL FILE, and CLOSE FILE statements.
- You can pass a FILE variable as a parameter to a routine.
- Several built-in routines require a FILE variable as a parameter, such as BYTES\_LEFT, CLR\_IO\_STAT, GET\_FILE\_POS, IO\_STATUS, SET\_FILE\_POS.
- FILE variables have these restrictions:
  - FILE variables must be static variables.
  - FILE variables are never saved.
  - FILE variables cannot be function return values.
  - FILE types are not allowed in structures, but are allowed in arrays.

- No other use of this variable data type, including assignment to one another, is permitted.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

## A.6.2 **FILE\_LIST** Built-In Procedure

**Purpose:** Generates a list of files with the specified name and type on the specified device.

**Syntax:** FILE\_LIST(file\_spec, n\_skip, format, ary\_nam, n\_files, status)

Input/Output Parameters:

[in] file\_spec : STRING  
[in] n\_skip : INTEGER  
[in] format : INTEGER  
[out] ary\_nam : ARRAY of STRING  
[out] n\_files : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : BYNAM

### Details:

- file\_spec specifies the device, name, and type of the list of files to be found. file\_spec can be specified using the wildcard (\*) character.
  - n\_skip is used when more files exist than the declared length of ary\_nam. Set n\_skip to 0 the first time you use FILE\_LIST. If ary\_nam is completely filled with variable names, copy the array to another ARRAY of STRINGS and execute the FILE\_LIST again with n\_skip equal to n\_files. The second call to FILE\_LIST will skip the files found in the first pass and only locate the remaining files.
  - format specifies the format of the file name and file type. The following values are valid for format:
    - 1 file\_name only, no blanks
    - 2 file\_type only, no blanks
    - 3 file\_name.file\_type, no blanks
    - 4 filename.ext size date time
  - The total length is 40 characters:
    - The file\_name starts with character 1.
    - The file\_type (extension) starts with character 10.
    - The size starts with character 21.
    - The date starts with character 26.
    - The time starts with character 36.
- Date and time are returned only if the device supports time stamping; otherwise just the filename.ext size is stored.
- ary\_nam is an ARRAY of STRINGS to store the file names. If the string length of ary\_nam is not large enough to store the formatted information, an error will be returned.
  - n\_files is the number of files stored in ary\_name.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.22.29, VAR\\_LIST Built-In Procedure](#), [Section A.16.23, PROG\\_LIST Built-In Procedure](#)

**Example:** Refer to [Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS](#) (LIST\_EX.KL), for a detailed program example.

## A.6.3 FOR...ENDFOR Statement

---

**Purpose:** Looping construct based on an INTEGER counter

**Syntax:** FOR count = initial || TO | DOWNT0 || final

DO {stmnt} ENDFOR

where:

[in]count : INTEGER variable

[in]initial : INTEGER expression

[in]final : INTEGER expression

[in]stmt : executable KAREL statement

### Details:

- Initially, count is set to the value of initial and final is evaluated. For each iteration, count is compared to final.
- If TO is used, count is incremented for each loop iteration.
- If DOWNT0 is used, count is decremented for each loop iteration.
- If count is greater than final using TO, stmt is never executed.
- If count is less than final using DOWNT0, stmt is never executed on the first iteration.
- If the comparison does not fail on the first iteration, the FOR loop will be executed for the number of times that equals ABS( final - initial ) + 1.
- If final = initial, the loop is executed once.
- initial is evaluated prior to entering the loop. Therefore, changing the values of initial and final during loop execution has no effect on the number of iterations performed.
- The value of count on exit from the loop is uninitialized.
- Never issue a GO TO statement in a FOR loop. If a GO TO statement causes the program to exit a FOR loop, the program might be aborted with a Run time stack overflow error.
- Never include a GO TO label in a FOR loop. Entering a FOR loop by a GO TO statement usually causes the program to be aborted with a Run time stack underflow error when the ENDFOR statement is encountered.
- The program will not be translated if *count* is a system variable or ARRAY element.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#), for additional syntax information.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES](#) (CPY\_PTH.KL)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM](#) (PTH\_MOVE.KL)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS](#) (LIST\_EX.KL)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES  
\(CHG\\_DATA.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING  
\(DOUT\\_EX.KL\)](#)

## A.6.4 **FORCE\_LINK** Built-In Procedure

**Purpose:** Forces the display of custom web pages or generic links.

**Syntax:** FORCE\_LINK(pane\_id, url)

Input/Output Parameters:

[in] pane\_id : INTEGER

[in] url : STRING

%ENVIRONMENT Group : PBCORE

### Details:

- pane\_id specifies the pane and should be one of the following predefined constants:
  - tp\_panel for primary iPendant pane
  - tp2\_panel for dual iPendant pane
  - tp3\_panel for third iPendant pane
  - 4-9 are reserved for panes during remote connections
- url is an acronym for Uniform Resource Location and is an address to a resource on the Internet or a custom web page on the controller. It can be any valid URL location. Linking URLs are also provided to force configuration and other special uses.

### Examples:

Force fr:pw\_op1.stm into primary pane of iPendant:

```
FORCE_LINK(tp_panel, '/fr/pw_op1.stm')
```

Force fr:pw\_op1.stm into the currently focused frame of iPendant:

```
FORCE_LINK(tp_panel, 'current=/fr/pw_op1.stm')
```

Force other robot home pages into second and third pane of iPendant:

```
FORCE_LINK(tp2_panel, 'http://robot05')
FORCE_LINK(tp3_panel, 'http://192.168.0.6')
```

A special link can be used to force the page in the appropriate pane to be refreshed:

```
FORCE_LINK(tp_panel, 'refresh=prim')
FORCE_LINK(tp_panel, 'refresh=dual')
FORCE_LINK(tp_panel, 'refresh=third')
```

### Linking URLs:

Generic linking provides the following key capabilities:

- Specification of any menu page in the system as a URL.
- Specification of any program edit as a URL.
- Specification of any of the three *panes* on *iPendant* as the destination for a link.
- Specification of links into multiple panes from a single complex link.
- Specification explicitly or implicitly of any display configuration.

Linking URLs can be complex, but you can easily create your own. Bring up the *iPendant* into the configuration you require. Press **SHIFT-DISP** and select **User Views. Select – Add Current** – and note the number it used to set it. For example, 1 Alarm Log | Registers

Look at `$UI_USERVIEW[1]` or whatever number it used.

```
$CONFIG=DOUBLE
$PRIM=menupage,18,1
$DUAL=menupage,23,1
```

Separate fields with & and remove the \$ from the field names. The URL built from `$UI_USERVIEW[1]` would be:

```
FORCE_LINK(tp_panel,
'CONFIG=DOUBLE&PRIM=menupage,18,1&DUAL=menupage,23,1')
```

It is case insensitive.

If you just want to change the configuration, you can remove the other parameters. Please refer the *iPendant Customization Guide (MARUUIPCU02201E)* or *iPendant Customization OPERATOR'S MANUAL (B-83594EN)*, Generic Linking Detailed Information, for more information on building linking URLs.

## A.6.5 FORCE\_SPMENU Built-In Procedure

**Purpose:** Forces the display of the specified menu

**Syntax:** `FORCE_SPMENU(device_code, spmenu_id, screen_no)`

Input/Output Parameters:

[in] `device_code` : INTEGER

[in] `spmenu_id` : INTEGER

[in] `screen_no` : INTEGER

%ENVIRONMENT Group : PBCORE

### Details:

- `device_code` specifies the device and should be one of the following predefined constants:
  - `tp_panel` Teach pendant device
  - `crt_panel` CRT device
- `spmenu_id` and `screen_no` specify the menu to force. The predefined constants beginning with `SPI_` define the `spmenu_id` and the predefined constants beginning with `SCR_` define the `screen_no`. If no `SCR_` is listed, use 1.
  - `SPI_TPHINTS` — UTILITIES Hints

- SPI\_TPPRGADJ — **UTILITIES Prog Adjust**
- SPI TPMIRROR — **UTILITIES Mirror Image**
- SPI\_TPSHIFT — **UTILITIES Program Shift**
- SPI\_TPTSTRUN — **TEST CYCLE**
- SPI\_TPMANUAL, SCR\_MACMAN — **MANUAL Macros**
- SPI\_TPOTREL — **MANUAL OT Release**
- SPI\_TPALARMS, SCR\_ALM\_ALL — **ALARM Alarm Log**
- SPI\_TPALARMS, SCR\_ALM\_MOT — **ALARM Motion Log**
- SPI\_TPALARMS, SCR\_ALM\_SYS — **ALARM System Log**
- SPI\_TPALARMS, SCR\_ALM\_APPL — **ALARM Appl Log**
- SPI\_TPDIGIO — **I/O Digital**
- SPI\_TPANAIO — **I/O Analog**
- SPI\_TPGRPIO — **I/O Group**
- SPI\_TPROBIO — **I/O Robot**
- SPI\_TPUOPIO — **I/O UOP**
- SPI\_TPSOPIO — **I/O SOP**
- SPI\_TPPLCIO — **I/O PLC**
- SPI\_TPSETGEN — **SETUP General**
- SPI\_TPFRAM — **SETUP Frames**
- SPI TPPORT — **SETUP Port Init**
- SPI\_TPMACRO, SCR\_MACSETUP — **SETUP Macro**
- SPI\_TPREFPOS — **SETUP Ref Position**
- SPI\_TPPWORD — **SETUP Passwords**
- SPI\_TPHCOMM — **SETUP Host Comm**
- SPI\_TPSYRSR — **SETUP RSR/PNS**
- SPI\_TPFILS — **FILE**
- SPI\_TPSTATUS, SCR\_AXIS — **STATUS Axis**
- SPI\_TPMEMORY — **STATUS Memory**
- SPI\_TPVERSN — **STATUS Version ID**
- SPI\_TPPRGSTS — **STATUS Program**
- SPI\_TPSFTY — **STATUS Safety Signals**
- SPI\_TPUSER — **USER**
- SPI\_TPSELECT — **SELECT**
- SPI\_TPTCH — **EDIT**
- SPI\_TPREGIS, SCR\_NUMREG — **DATA Registers**
- SPI\_SFMPREG, SCR\_POSREG — **DATA Position Reg**
- SPI\_TPSYSV, SCR\_NUMVAR — **DATA KAREL Vars**
- SPI\_TPSYSV, SCR\_POSVAR — **DATA KAREL Posns**
- SPI\_TPOPOSN — **POSITION**
- SPI\_TPSYSV, SCR\_CLOCK — **SYSTEM Clock**
- SPI\_TPSYSV, SCR\_SYSVAR — **SYSTEM Variables**
- SPI TPMASCAL — **SYSTEM Master/Cal**
- SPI\_TPBRKCTR — **SYSTEM Brake Cntrl**
- SPI\_TPAXLM — **SYSTEM Axis Limits**
- SPI\_CRTKCL, SCR\_KCL — **KCL> (crt\_panel only)**
- SPI\_CRTKCL, SCR\_CRT — **KAREL EDITOR (crt\_panel only)**

- SPI\_TPUSER2 — Menu for form/table managers

**See Also:** [Section A.1.7, ACT\\_SCREEN Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.4, STANDARD ROUTINES \(ROUT\\_EX.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLISTEG.UTX\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.6.6 FORMAT\_DEV Built-In Procedure

---

**Purpose:** Deletes any existing information and records a directory and other internal information on the specified device.

**Syntax:** FORMAT\_DEV(device, volume\_name, nowait\_sw, status)

Input/Output Parameters:

[in] device : STRING

[in] volume\_name : STRING

[in] nowait\_sw : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- device specifies the device to initialize.
- volume\_name acts as a label for a particular unit of storage media. volume\_name can be a maximum of 11 characters and will be truncated to 11 characters if more are specified.
- If nowait\_sw is TRUE, execution of the program continues while the command is executing. If it is FALSE, the program stops, including condition handlers, until the operation is complete. If you have time critical condition handlers in the program, put them in another program that executes as a separate task.

### NOTE

nowait\_sw is not available in this release and should be set to FALSE.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.13.4, MOUNT\\_DEV Built-In Procedure](#), [Section A.4.26, DISMOUNT\\_DEV Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

## A.6.7 FRAME Built-In Function

**Purpose:** Returns a frame with a POSITION data type representing the transformation to the coordinate frame specified by three (or four) POSITION arguments.

**Syntax:** FRAME(pos1, pos2, pos3 <,pos4>)

Function Return Type : POSITION

Input/Output Parameters:

[in]pos1 : POSITION

[in]pos2 : POSITION

[in]pos3 : POSITION

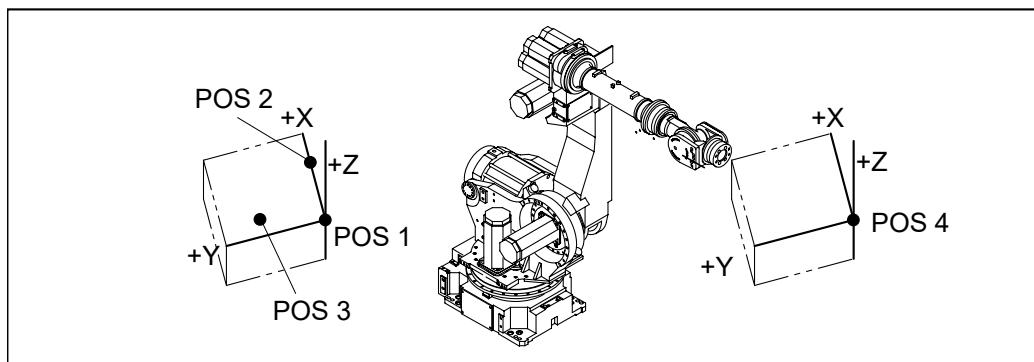
[in]pos4 : POSITION

%ENVIRONMENT Group : SYSTEM

### Details:

- The returned value is computed as follows:
  - pos1 is assumed to be the origin unless a pos4 argument is supplied. See [Figure A.6.7 \(a\)](#).
  - If pos4 is supplied, the origin is shifted to pos4, and the new coordinate frame retains the same orientation in space as the first coordinate frame. See [Figure A.6.7 \(a\)](#).
  - The x-axis is parallel to a line from pos1 to pos2.
  - The xy-plane is defined to be that plane containing pos1, pos2, and pos3, with pos3 in the positive half of the plane.
  - The y-axis is perpendicular to the x-axis and in the xy-plane.
  - The z-axis is through pos1 and perpendicular to the xy-plane. The positive direction is determined by the right hand rule.
  - The configuration of the result is set to that of pos1, or pos4 if it is supplied.
- pos1 and pos2 arguments must be at least 10 millimeters apart and pos3 must be at least 10 millimeters away from the line connecting pos1 and pos2.

If either condition is not met, the program is paused with an error.



**Figure A.6.7 (a) FRAME Built-In Function**

**Example:** The following example allows the operator to set a frame to a pallet so that a palletizing routine will be able to move the TCP along the x, y, z direction in the pallet's coordinate frame.

```
WRITE('Teach corner_1, corner_2, corner_3',CR)
RELEASE --Allows operator to turn on teach pendant
```

```
--and teach positions
ATTACH --Returns motion control to program
$UFRAME = FRAME (corner_1, corner_2, corner_3)
```

**Figure A.6.7 (b) FRAME Built-In Function**

## A.6.8 FROM Clause

---

**Purpose:** Indicates a variable or routine that is external to the program, allowing data and/or routines to be shared among programs

**Syntax:** FROM prog\_name

where:

prog\_name : any KAREL program identifier

**Details:**

- The FROM clause can be part of a type, variable, or routine declaration.
- The type, variable, or routine belongs to the program specified by prog\_name.
- In a FROM clause, prog\_name can be the name of any program, including the program in which the type, variable, or routine is declared.
- If the FROM clause is used in a routine declaration and is called during program execution, the body of the declaration must appear in the specified program and that program must be loaded.
- The FROM clause cannot be used when declaring variables in the declaration section of a routine.

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.7 - G - KAREL LANGUAGE DESCRIPTION

---

### A.7.1 GET\_ATTR\_PRG Built-In Procedure

---

**Purpose:** Gets attribute data from the specified teach pendant or KAREL program

**Syntax:** GET\_ATTR\_PRG(program\_name, attr\_number, int\_value, string\_value, status)

Input/Output Parameters:

[in] program\_name : STRING

```
[in] attr_number : INTEGER
[out] int_value : INTEGER
[out] string_value : STRING
[out] status : INTEGER
%ENVIRONMENT Group : PBCORE
```

**Details:**

- program\_name specifies the program from which to get attribute.
- attr\_number is the attribute whose value is to be returned. The following attributes are valid:
  - AT\_PROG\_TYPE : Program type
  - AT\_PROG\_NAME : Program name (String[12])
  - AT\_OWNER : Owner (String[8])
  - AT\_COMMENT : Comment (String[16])
  - AT\_PROG\_SIZE : Size of program
  - AT\_ALLC\_SIZE : Size of allocated memory
  - AT\_NUM\_LINE : Number of lines
  - AT\_CRE\_TIME : Created (loaded) time
  - AT\_MDFY\_TIME : Modified time
  - AT\_SRC\_NAME : Source file ( or original file ) name (String[128])
  - AT\_SRC\_VRSN : Source file versionA
  - AT\_DEF\_GROUP : Default motion group mask (for task attribute). See [Table A.12.4](#).
  - AT\_PROTECT : Protection code:
    - 1 : Protection OFF
    - 2 : Protection ON
  - AT\_STORAGE : Storage type:
    - TPSTOR\_CMOS
    - TPSTOR\_SHADOW
    - TPSTOR\_FILE
    - TPSTOR\_SHOD
  - AT\_STK\_SIZE : Stack size (for task attribute)
  - AT\_TASK\_PRI : Task priority (for task attribute)
  - AT\_DURATION : Time slice duration (for task attribute)
  - AT\_BUSY\_OFF : Busy lamp off (for task attribute)
  - AT\_IGNR\_ABRT : Ignore abort request (for task attribute)
  - AT\_IGNR\_PAUS : Ignore pause request (for task attribute)
  - AT\_CONTROL : Control code (for task attribute)
- The program type returned for AT\_PROG\_TYPE will be one of the following constants:
  - PT\_KRLPRG : KAREL program
  - PT\_MNE\_UNDEF : Teach pendant program of undefined sub type
  - PT\_MNE\_JOB : Teach pendant job
  - PT\_MNE\_PROC : Teach pendant process
  - PT\_MNE\_MACRO : Teach pendant macro
- If the attribute data is a number, it is returned in int\_value and string\_value is not modified.
- If the attribute data is a string, it is returned in string\_value and int\_value is not modified.

- status explains the status of the attempted operation. If it is not equal to 0, then an error has occurred. Some of the errors which could occur are:
  - 7073** The program specified in program\_name does not exist
  - 17027** string\_value is not large enough to contain the attribute string. The value has been truncated to fit.
  - 17033** attr\_number has an illegal value

**See Also:** [Section A.19.9, SET\\_ATTR\\_PRG Built-In Procedure](#), [Section A.7.24, GET\\_TSK\\_INFO Built-In Procedure](#), [Section A.19.36, SET\\_TSK\\_ATTR Built-In Procedure](#)

## A.7.2 GET\_FILE\_POS Built-In Function

---

**Purpose:** Returns the current file position (where the next READ or WRITE operation will take place) in the specified file

**Syntax:** GET\_FILE\_POS(file\_id)

Function Return Type : INTEGER

Input/Output Parameters:

[in] file\_id : FILE

%ENVIRONMENT Group : FLBT

**Details:**

- GET\_FILE\_POS returns the number of bytes before the next byte to be read or written in the file.
- Line terminators are counted in the value returned.
- The file associated with file\_id must be open. Otherwise, the program is aborted with an error.
- If the file associated with file\_id is open for read-only, it cannot be on the FROM or RAM disks as a compressed file.

 **WARNING**

GET\_FILE\_POS is only supported for files opened on the RAM Disk device. Do not use GET\_FILE\_POS on another device; otherwise, you could injure personnel and damage equipment.

**Example:** The following example opens the filepos.dt data file, stores the positions in my\_path in the file, and builds a directory to access them.

```
OPEN FILE file_id ('RW', 'filepos.dt')
FOR i = 1 TO PATH_LEN(my_path) DO
  temp_pos = my_path[i].node_pos
  pos_dir[i] = GET_FILE_POS(file_id)
  WRITE file_id (temp_pos)
ENDFOR
```

**Figure A.7.2 GET\_FILE\_POS Built-In Function**

## A.7.3 GET\_JPOS\_REG Built-In Function

**Purpose:** Gets a JOINTPOS value from the specified register

**Syntax:** GET\_JPOS\_REG(register\_no, status <,group\_no>)

Function Return Type : REGOPE

Input/Output Parameters:

[in] register\_no : INTEGER

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register to get.
- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- GET\_JPOS\_REG returns the position in JOINTPOS format. Use POS\_REG\_TYPE to determine the position representation.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.12, GET\\_POS\\_REG Built-In Function](#) , [Section A.19.16, SET\\_JPOS\\_REG Built-In Procedure](#) , [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#)

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#) for a detailed program example.

## A.7.4 GET\_JPOS\_TPE Built-In Function

**Purpose:** Gets a JOINTPOS value from the specified position in the specified teach pendant program

**Syntax:** GET\_JPOS\_TPE(open\_id, position\_no, status <, group\_no>)

Function Return Type : JOINTPOS

Input/Output Parameters:

[in] open\_id : INTEGER

[in] position\_no : INTEGER

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the program to get.
- If group\_no is omitted, the default group for the program is assumed.

- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- No conversion is done for the position representation. The position data must be in JOINTPOS format. If the stored position is not in JOINTPOS, an error status is returned. Use GET\_POS\_TYP to get the position representation.
- If the specified position in the program is uninitialized, the returned JOINTPOS value is uninitialized and the status is set to **17038**, Uninitialized TPE position.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**See Also:** [Section A.19.17, SET\\_JPOS\\_TPE Built-In Procedure](#) , [Section A.7.13, GET\\_POS\\_TPE Built-In Function](#) , [Section A.19.27, SET\\_POS\\_TPE Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TPE.KL\)](#), for a detailed program example.

## A.7.5 GET\_PORT\_ASG Built-in Procedure

**Purpose:** Allows a KAREL program to determine the physical port(s) to which a specified logical port is assigned.

**Syntax:** GET\_PORT\_ASG(log\_port\_type, log\_port\_no, rack\_no, slot\_no, phy\_port\_type, phy\_port\_no, n\_ports, status)

Input/Output Parameters:

[in] log\_port\_type : INTEGER  
 [in] log\_port\_no : INTEGER  
 [out] rack\_no : INTEGER  
 [out] slot\_no : INTEGER  
 [out] phy\_port\_type : INTEGER  
 [out] phy\_port\_no : INTEGER  
 [out] n\_ports : INTEGER  
 [out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

### Details:

- log\_port\_type specifies the code for the type of port whose assignment is being accessed. Codes are defined in FR:KLIOTYPS.KL.
- log\_port\_no specifies the number of the port whose assignment is being accessed.
- rack\_no is returned with the rack containing the port module. For process I/O boards, memory-image, and dummy ports, this is zero; for Allen-Bradley ports, this is 16.
- phy\_port\_type is returned with the type of port assigned to. Often this will be the same as log\_port\_type. Exceptions are if log\_port\_type is a group type (io\_gpin or io\_gpout) or a port is assigned to memory-image or dummy ports.
- phy\_port\_no is returned with the number of the port assigned to. If log\_port\_type is a group, this is the port number for the least-significant bit of the group.

- n\_ports is returned with the number of physical ports assigned to the logical port. This will be 1 in all cases except when log\_port\_type is a group type. In this case, n\_ports indicates the number of bits in the group.
- status is returned with zero if the parameters are valid and the specified port is assigned. Otherwise, it is returned with an error code.

**Example:** The following example returns to the caller the module rack and slot number, port\_number, and number of bits assigned to a specified group input port. A BOOLEAN is returned indicating whether the port is assigned to a DIN port. If the port is not assigned, a non-zero status is returned.

```

PROGRAM getasgprog
  %ENVIRONMENT IOSETUP
  %INCLUDE FR:\kliotyps
    ROUTINE get_gin_asg(gin_port_no: INTEGER;
                         rack_no: INTEGER;
                         slot_no: INTEGER;
                         frst_port_no: INTEGER;
                         n_ports: INTEGER;
                         asgd_to_din: BOOLEAN): INTEGER
    VAR
      phy_port_typ: INTEGER
      status: INTEGER

      BEGIN
        GET_PORT_ASG(io_gpin, gin_port_no, rack_no, slot_no,
                     phy_port_typ, frst_port_no, n_ports, status)
        IF status <> 0 THEN
          RETURN (status)
        ENDIF
        asgd_to_din = (phy_port_typ = io_din)
      END get_gin_asg
    BEGIN
  END getasgprog

```

Figure A.7.5 GET\_PORT\_ASG Built-In Procedure

## A.7.6 GET\_PORT\_ATR Built-In Function

**Purpose:** Gets an attribute from the specified port

**Syntax:** GET\_PORT\_ATR(port\_id, atr\_type, atr\_value)

Function Return Type: INTEGER

Input/Output Parameters:

[in] port\_id : INTEGER  
 [in] atr\_type : INTEGER  
 [out] atr\_value : INTEGER

%ENVIRONMENT Group : FLBT

### Details:

- port\_id specifies which port is to be queried. Use one of the following predefined constants:
  - port\_1

- port\_2
- port\_3
- port\_4
- port\_5
- atr\_type specifies the attribute whose current setting is to be returned. Use one of the following predefined constants:
  - atr\_readahd : Read ahead buffer
  - atr\_baud : Baud rate
  - atr\_parity : Parity
  - atr\_sbts : Stop bit
  - atr\_dbts : Data length
  - atr\_xonoff : Xon/Xoff
  - atr\_eol : End of line
  - atr\_modem : Modem line
- atr\_value receives the current value for the specified attribute.
- GET\_PORT\_ATR returns the status of this action to the port.

**See Also:** [Section A.19.21, SET\\_PORT\\_ATR Built-In Function](#), [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#)

**Example:** The following example sets up the port to a desired configuration, if it is not already set to the specified configuration.

```
PROGRAM port_atr
%ENVIRONMENT FLBT
VAR
  stat: INTEGER
  atr_value: INTEGER
BEGIN
  -- sets read ahead buffer to desired value, if not already correct
  stat=GET_PORT_ATR(port_2,atr_readahd,atr_value)
  IF(atr_value <> 2) THEN
    stat=SET_PORT_ATR(port_2,atr_readahd,2) --set to 256 bytes
  ENDIF
  -- sets the baud rate to 9600, if not already set
  stat=GET_PORT_ATR(port_2,atr_baud,atr_value)
  IF(atr_value <> BAUD_9600) THEN
    stat=SET_PORT_ATR(port_2,atr_baud,baud_9600)
  ENDIF
  -- sets parity to even, if not already set
  stat=GET_PORT_ATR(port_2,atr_parity,atr_value)
  IF(atr_value <> PARITY_EVEN) THEN
    stat=SET_PORT_ATR(port_2,atr_parity,PARITY_EVEN)
  ENDIF
  -- sets the stop bit to 1, if not already set
  stat=GET_PORT_ATR(port_2,atr_sbts,atr_value)
  IF(atr_value <> SBITS_1) THEN
    stat=SET_PORT_ATR(port_2,atr_sbts,SBITS_1)
  ENDIF
  -- sets the data bit to 5, if not already set
  stat=GET_PORT_ATR(port_2,atr_dbts,atr_value)
  IF(atr_value <> DBITS_5) THEN
    stat=SET_PORT_ATR(port_2,atr_dbts,DBITS_5)
  ENDIF
  -- sets xonoff to not used, if not already set
  stat=GET_PORT_ATR(port_2,atr_xonoff,atr_value)
```

```
IF(atr_value <> xf_not_used) THEN
    stat=SET_PORT_ATR(port_2,atr_xonoff,xf_not_used)
ENDIF
-- sets end of line marker, if not already set
stat=GET_PORT_ATR(port_2,atr_eol,atr_value)
IF(atr_value <> 65) THEN
    stat=SET_PORT_ATR(port_2,atr_eol,65)
ENDIF
END port_attr
```

**Figure A.7.6 GET\_PORT\_ATR Built-In Function**

## A.7.7 GET\_PORT\_CMT Built-In Procedure

**Purpose:** Allows a KAREL program to determine the comment that is set for a specified logical port

**Syntax:** GET\_PORT\_CMT(port\_type, port\_no, comment\_str, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[out] comment\_str : STRING

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port whose comment is being returned. Codes are defined in FR:KLIOTYPS.KL.
- port\_no specifies the port number whose comment is being set.
- comment\_str is returned with the comment for the specified port. This should be declared as a STRING with a length of at least 16 characters.
- status is returned with zero if the parameters are valid and the comment is returned for the specified port.

**See Also:** [Section A.7.10, GET\\_PORT\\_VAL Built-In Procedure](#) , [Section A.7.8, GET\\_PORT\\_MOD Built-In Procedure](#) , [Section A.19.22, SET\\_PORT\\_CMT Built-In Procedure](#) , [Section A.19.25, SET\\_PORT\\_VAL Built-In Procedure](#) , [Section A.19.23, SET\\_PORT\\_MOD Built-In Procedure](#)

## A.7.8 GET\_PORT\_MOD Built-In Procedure

**Purpose:** Allows a KAREL program to determine what special port modes are set for a specified logical port

**Syntax:** GET\_PORT\_MOD(port\_type, port\_no, mode\_mask, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[out] mode\_mask : INTEGER  
 [out] status : INTEGER  
 %ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port whose mode is being returned. Codes are defined in FR:KLIOTYPS.KL.
- port\_no specifies the port number whose mode is being set.
- mode\_mask is returned with a mask specifying which modes are turned on. The following modes are defined:
  - 1 : reverse mode

Sense of the port is reversed; if the port is set to TRUE, the physical output is set to FALSE. If the port is set to FALSE, the physical output is set to TRUE. If a physical input is TRUE, when the port is read, FALSE is returned. If a physical input is FALSE, when the port is read, TRUE is returned.

- 2 : complementary mode

The logical port is assigned to two physical ports whose values are complementary. In this case, port\_no must be an odd number. If port n is set to TRUE, then port n is set to TRUE and port n + 1 is set to FALSE. If port n is set to FALSE, then port n is set to FALSE and port n + 1 is set to TRUE. This is effective only for output ports.

- status is returned with zero if the parameters are valid and the specified mode is returned for the specified port.

**Example:** The following example gets the mode(s) for a specified port.

```
PROGRAM getmodprog
%ENVIRONMENT IOSETUP
%INCLUDE FR:\kliotypes
ROUTINE get_mode( port_type: INTEGER;
                  port_no:   INTEGER;
                  reverse:   BOOLEAN;
                  complementary: BOOLEAN) : INTEGER
VAR
  mode:   INTEGER
  status: INTEGER
BEGIN
  GET_PORT_MOD(port_type, port_no, mode, status)
  IF (status <>0) THEN
    RETURN (status)
  ENDIF
  IF (mode AND 1) <> 0 THEN
    reverse = TRUE
  ELSE
    reverse = FALSE
  ENDIF
  IF (mode AND 2) <> 0 THEN
    complementary = TRUE
  ELSE
    complementary = FALSE
  ENDIF
  RETURN (status)
END_get_mode
```

```
BEGIN
END getmodprog
```

**Figure A.7.8 GET\_PORT\_MOD\_Built-In Procedure**

## A.7.9 GET\_PORT\_SIM Built-In Procedure

**Purpose:** Gets port simulation status

**Syntax:** GET\_PORT\_SIM(port\_type, port\_no, simulated, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[out] simulated : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port to get. Codes are defined in FRS:KLIOTYPS.KL.
- port\_no specifies the number of the port whose simulation status is being returned.
- simulated returns TRUE if the port is being simulated, FALSE otherwise.
- status is returned with zero if the port is valid.

**See Also:** [Section A.7.8, GET\\_PORT\\_MOD Built-In Procedure](#) , [Section A.19.24, SET\\_PORT\\_SIM Built-In Procedure](#) , [Section A.19.23, SET\\_PORT\\_MOD Built-In Procedure](#)

## A.7.10 GET\_PORT\_VAL Built-In Procedure

**Purpose:** Allows a KAREL program to determine the current value of a specified logical port

**Syntax:** GET\_PORT\_VAL(port\_type, port\_no, value, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[out] value : STRING

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port whose comment is being returned. Codes are defined in FR:KLIOTYPS.KL.
- port\_no specifies the port number whose comment is being set.
- value is returned with the current value (status) of the specified port. For BOOLEAN port types, (such as DIN), this will be 0 = OFF, or 1 = ON.

- status is returned with zero if the parameters are valid and the comment is returned for the specified port.

**See Also:** [Section A.7.7, GET\\_PORT\\_CMT Built-In Procedure](#), [Section A.7.8, GET\\_PORT\\_MOD Built-In Procedure](#), [Section A.19.22, SET\\_PORT\\_CMT Built-In Procedure](#), [Section A.19.25, SET\\_PORT\\_VAL Built-In Procedure](#), [Section A.19.23, SET\\_PORT\\_MOD Built-In Procedure](#)

## A.7.11 GET\_POS\_FRM Built-In Procedure

---

**Purpose:** Gets the uframe number and utool number of the specified position in the specified teach pendant program.

**Syntax:** GET\_POS\_FRM(open\_id, position\_no, gnum, ufram\_no, utool\_no, status)

Input/Output Parameters:

[in] open\_id : INTEGER

[in] position\_no : INTEGER

[in] gnum : INTEGER

[out] ufram\_no : INTEGER

[out] utool\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the teach pendant program.
- gnum specifies the group number of position.
- ufram\_no is returned with the frame number of position\_no.
- utool\_no is returned with the tool number of position\_no.
- If the specified position, position\_no, is uninitialized, the status is set to **17038, Uninitialized TPE position**.
- status indicates the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.14, GET\\_POS\\_TYP Built-In Procedure](#), [Section A.3.7, CHECK\\_EPOS Built-In Procedure](#)

## A.7.12 GET\_POS\_REG Built-In Function

---

**Purpose:** Gets an XYZWPR value from the specified register

**Syntax:** GET\_POS\_REG(register\_no, status <,group\_no>)

Function Return Type: XYZWPREXT

Input/Output Parameters:

[in] register\_no : INTEGER

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register to get.
- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- GET\_POS\_REG returns the position in XYZWPREXT format. Use POS\_REG\_TYPE to determine the position representation.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.3, GET\\_JPOS\\_REG Built-In Function](#) , [Section A.19.16, SET\\_JPOS\\_REG Built-In Procedure](#) , [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#) , [Section A.7.17, GET\\_REG Built-In Procedure](#)

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## **A.7.13 GET\_POS\_TPE Built-In Function**

**Purpose:** Gets an XYZWPREXT value from the specified position in the specified teach pendant program

**Syntax:** GET\_POS\_TPE(open\_id, position\_no, status <, group\_no>)

Function Return Type: XYZWPREXT

Input/Output Parameters:

[in] open\_id : INTEGER

[in] position\_no : INTEGER

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the program to get.
- No conversion is done for the position representation. The positional data must be in XYZWPR or XYZWPREXT, otherwise, an error status is returned. Use GET\_POS\_TYP to get the position representation.
- If the specified position in the program is uninitialized, the returned XYZWPR value is uninitialized and status is set to **17038, Uninitialized TPE Position**.
- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**See Also:** [Section A.7.4, GET\\_JPOS\\_TPE Built-In Function](#), [Section A.19.17, SET\\_JPOS\\_TPE Built-In Procedure](#), [Section A.19.27, SET\\_POS\\_TPE Built-In Procedure](#), [Section A.7.14, GET\\_POS\\_TYP Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.7.14 GET\_POS\_TYP Built-In Procedure

---

**Purpose:** Gets the position representation of the specified position in the specified teach pendant program

**Syntax:** GET\_POS\_TYP(open\_id, position\_no, group\_no, posn\_typ, num\_axs, status)

Input/Output Parameters:

- [in] open\_id : INTEGER
- [in] position\_no : INTEGER
- [in] group\_no : INTEGER
- [out] posn\_typ : INTEGER
- [out] num\_axs : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the program.
- group\_no specifies the group number.
- Position type is returned by posn\_typ. posn\_typ is defined as follows:
  - 2 : XYZWPR
  - 6 : XYZWPREXT
  - 9 : JOINTPOS
- If it is in joint position, the number of the axis in the representation is returned by num\_axs.
- If the specified position in the program is uninitialized, then a status is set to **17038**, Uninitialized TPE Position.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_T.KL\)](#), for a detailed program example.

## A.7.15 GET\_PREG\_CMT Built-In-Procedure

---

**Purpose:** To retrieve the comment information of a KAREL position register based on a given register number.

**Syntax:** GET\_PREG\_CMT (register\_no, comment\_string, status)

Input/Output Parameters:

[in] register\_no: INTEGER  
[out] comment\_string: STRING  
[out] status: INTEGER  
%ENVIRONMENT group: REGOPE

**Details:**

- register\_no specifies which position register to retrieve the comments from. The comment of the given position register is returned in the comment\_string.

## A.7.16 GET\_QUEUE Built-In Procedure

**Purpose:** Retrieves the specified oldest entry from a queue

**Syntax:** GET\_QUEUE(queue, queue\_data, value, status, sequence\_no)

Input/Output Parameters:

[in,out] queue\_t : QUEUE\_TYPE  
[in,out] queue\_data : ARRAY OF INTEGER  
[out] value : INTEGER  
[out] sequence\_no : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : PBQMGR

**Details:**

- queue\_t specifies the queue variable for the queue from which the value is to be obtained.
- queue\_data specifies the array variable with the queue data.
- value is returned with the oldest entry obtained from the queue.
- sequence\_no is returned with the sequence number of the returned entry.
- status is returned with the zero if an entry is successfully obtained from the queue. Otherwise, a value of **61002**, Queue is empty, is returned.

**See Also:** [Section A.13.2, MODIFY\\_QUEUE Built-In Procedure](#) , [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

**Example:** In the following example the routine get\_nxt\_err returns the oldest entry from the error queue, or zero if the queue is empty.

```
PROGRAM get_queue_x
  %environment PBQMGR
  VAR
    error_queue FROM global_vars: QUEUE_TYPE
    error_data FROM global_vars: ARRAY[100] OF INTEGER

  ROUTINE get_nxt_err: INTEGER
  VAR
    status: INTEGER
    value: INTEGER
    sequence_no: INTEGER
```

```

BEGIN
GET_QUEUE(error_queue, error_data, value, sequence_no, status)
IF (status = 0) THEN
    RETURN (value)
ELSE
    RETURN (0)
ENDIF
END get_nxt_err
BEGIN
END get_queue_x

```

**Figure A.7.16 GET\_QUEUE Built-In Procedure**

## A.7.17 GET\_REG Built-In Procedure

**Purpose:** Gets an INTEGER or REAL value from the specified register

**Syntax:** GET\_REG(register\_no, real\_flag, int\_value, real\_value, status)

Input/Output Parameters:

- [in] register\_no : INTEGER
- [out] real\_flag : BOOLEAN
- [out] int\_value : INTEGER
- [out] real\_value : REAL
- [out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the register to get.
- real\_flag is set to TRUE and real\_value to the register content if the specified register has a real value. Otherwise, real\_flag is set to FALSE and int\_value is set to the contents of the register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## A.7.18 GET\_REG\_CMT Built-In Procedure

**Purpose:** To retrieve the comment information of a KAREL register based on a given register number.

**Syntax:** GET\_REG\_CMT (register\_no, comment\_string, status)

Input/Output Parameters:

- [in] register\_no: INTEGER
- [out] comment\_string: STRING
- [out] status: INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies which register to retrieve the comments from. The comment of the given register is returned in comment\_string.

## A.7.19 GET\_SREG\_CMT Built-In Procedure

---

**Purpose:** Gets the comment from the specified string register.

**Syntax:** GET\_SREG\_CMT(register\_no, comment, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[out] comment : STRING[254]

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the string register to get.
- comment contains the comment of the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.20, GET\\_STR\\_REG Built-In Procedure](#), [Section A.19.32, SET\\_STR\\_REG Built-in Procedure](#), [Section A.19.31, SET\\_SREG\\_CMT Built-in Procedure](#)

## A.7.20 GET\_STR\_REG Built-In Procedure

---

**Purpose:** Gets the value from the specified string register.

**Syntax:** GET\_STR\_REG(register\_no, value, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[out] value : STRING[254]

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the string register to get.
- value contains the value of the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.19, GET\\_SREG\\_CMT Built-In Procedure](#), [Section A.19.32, SET\\_STR\\_REG Built-in Procedure](#), [Section A.19.31, SET\\_SREG\\_CMT Built-in Procedure](#)

## A.7.21 GET\_TIME Built-In Procedure

---

**Purpose:** Retrieves the current time (in integer representation) from within the KAREL system

**Syntax:** GET\_TIME(i)

Input/Output Parameters:

[out] i : INTEGER

%ENVIRONMENT Group : TIM

**Details:**

- i holds the INTEGER representation of the current time stored in the KAREL system. This value is represented in 32-bit INTEGER format as follows:

**Table A.7.21 INTEGER Representation of Current Time**

31–25	24–21	20–16
year	month	day
15–11	10–5	4–0
hour	minute	second

- The contents of the individual fields are as follows:

– DATE:

- Bits 31-25 — Year since 1980
- Bits 24-21 — Month (1-12)
- Bits 20-16 — Day of the month

– TIME:

- Bits 15-11 — Number of hours (0-23)
- Bits 10-5 — Number of minutes (0-59)
- Bits 4-0 — Number of 2-second increments (0-29)

- INTEGER values can be compared to determine if one time is more recent than another.
- Use the CNV\_TIME\_STR built-in procedure to convert the INTEGER into the DD-MMM-YYY HH:MM:SS STRING format.

**See Also:** [Section A.3.37, CNV\\_TIME\\_STR Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

---

## A.7.22 GET\_TPE\_CMT Built-in Procedure

---

**Purpose:** This built-in provides the ability for a KAREL program to read the comment associated with a specified position in a teach pendant program.

**Syntax:** GET\_TPE\_CMT(open\_id, pos\_no, comment, status)

Input/Output Parameters:

[in] open\_id : INTEGER

[in] pos\_no : INTEGER  
[out] comment : STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : TPE

**Details:**

- open\_id specifies the open\_id returned from a previous call to OPEN\_TPE.
- pos\_no specifies the number of the position in the TPP program to get a comment from.
- comment is associated with specified positions and is returned with a zero length string if the position has no comment. If the string variable is too short for the comment, an error is returned and the string is not changed.
- status indicates zero if the operation was successful, otherwise an error code will be displayed.

See Also: [Section A.19.34, SET\\_TPE\\_CMT Built-In Procedure](#)

## A.7.23 GET\_TPE\_PRM Built-in Procedure

**Purpose:** Gets the values of the parameters when parameters are passed in a TPE CALL or MACRO instruction.

**Syntax:** GET\_TPE\_PRM(param\_no, data\_type, int\_value, real\_value, str\_value, status)

Input/Output Parameters:

[in] param\_no : INTEGER  
[out] data\_type : INTEGER  
[out] int\_value : INTEGER  
[out] real\_value : REAL  
[out] str\_value : STRING  
[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- param\_no indicates the number of the parameter. There can be at most ten parameters.
- data\_type indicates the data type for the parameter, as follows:
  - 1 : INTEGER
  - 2 : REAL
  - 3 : STRING
- int\_value is the value of the parameter if the data\_type is INTEGER.
- real\_value is the value of the parameter if the data\_type is REAL.
- str\_value is the value of the parameter if the data\_type is STRING.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.
- If the parameter designated by param\_no does not exist, a status of **17042** is returned, which is the error message: **ROUT-042** **WARN** TPE parameters do not exist . If this error is returned, confirm the param\_no and the parameter in the CALL or MACRO command in the main TPE program.

**See Also:** Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (*B-83284EN*), for information on using parameters in teach pendant CALL or MACRO instructions.

**Example:** The following example shows the implementation of a macro (Send Event) with CALL parameters that are retrieved by a KAREL program that uses the GET\_TPE\_PRM built-in.

```

Macro table entry for the Send Event macro:
109 [Send Event] [SENDEVNT]--[ 0]
Teach pendant program, TEST1.TP, which uses the Send Event
macro:
1: ! Send Event 7
2: ! Wait for PC answer
3: ! Answer in REG 5
4: Send Event(7,1,5)
5: IF R[5]<9999,JMP LBL[10]
6: ! Error in macro
7: !
8: LBL[10]
Teach pendant program SENDEVNT.TP, which implements the
Send Event macro by calling the GESNDEVT KAREL
program and passing the CALL parameters from Send Event:
1: !Send Event Macro
2: CALL GESNDEVT(AR[1],AR[2],AR[3])
Snippet of the KAREL program GESNDEVT.KL, which gets the
parameter information using the GET_TPE_PRM
built-in:
PROGRAM GESNDEVT
. . .
BEGIN
-- Send Event(event_no [,wait_sw [,status_reg]] )
-- get parameter 1 (mandatory parameter)
Get_tpe_prm(1, data_type, event_no, real_value, string_value, status)
IF status<>0 THEN -- 17042 "ROUT-042 TPE parameters do not exist"
    POST_ERR(status, '', 0, er_abort)
ELSE
    IF data_type <> PARM_INTEGER THEN -- make sure parm is an
        -- integer
        POST_ERR(er_pceventer, '1', 0, er_abort)
    ELSE
        IF (event_no < MIN_EVENT) OR (event_no > MAX_EVENT) THEN
            POST_ERR(er_illevent, '', 0, er_abort)
        ENDIF
    ENDIF
ENDIF
-- get second parameter (optional)
Get_tpe_prm(2, data_type, wait_sw, real_value, string_value, status)
IF status<>0 THEN
    IF status = ER17042 THEN -- "ROUT-142 Parameter doesn't exist"
        wait_sw = 0 -- DEFAULT no wait
    ELSE
        POST_ERR(status, '', 0, er_warn) -- other error
    ENDIF
. . .

```

**Figure A.7.23 GET\_TPE\_PRM Built-In Procedure**

## A.7.24 GET\_TSK\_INFO Built-In Procedure

**Purpose:** Get the value of the specified task attribute

**Syntax:** GET\_TSK\_INFO(task\_name, task\_no, attribute, value\_int, value\_str, status)

Input/Output Parameters:

[in,out] task\_name : STRING

[in,out] task\_no : INTEGER

[in] attribute : INTEGER

[out] value\_int : INTEGER

[out] value\_str : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- task\_name is the name of the task of interest. task\_name is used as input only if task\_no is uninitialized or set to 0, otherwise, task\_name is considered an output parameter.
- task\_no is the task number of interest. If task\_no is uninitialized or set to 0, it is returned as an output parameter.
- attribute is the task attribute whose value is to be returned. It will be returned in value\_int unless otherwise specified. The following attributes are valid:
  - TSK\_HOLDCOND : Task hold conditions
  - TSK\_LINENUM : Current executing line number
  - TSK\_LOCKGRP : Locked group
  - TSK\_MCTL : Motion controlled groups
  - TSK\_NOABORT : Ignore abort request
  - TSK\_NOBUSY : Busy lamp off
  - TSK\_NOPAUSE : Ignore pause request
  - TSK\_NUMCLDS : Number of child tasks
  - TSK\_PARENT : Parent task number
  - TSK\_PAUSESFT : Pause on shift release
  - TSK\_PRIORITY : Task priority
  - TSK\_PROGNAME : Current program name returned in value\_str
  - TSK\_PROGTYPE : Program type - refer to description below
  - TSK\_ROUTNAME : Current routine name returned in value\_str
  - TSK\_STACK : Stack size
  - TSK\_STATUS : Task status — refer to description below
  - TSK\_STEP : Single step task
  - TSK\_TIMESLIC : Time slice duration in ms
  - TSK TPMOTION : TP motion enable
  - TSK\_TRACE : Trace enable
  - TSK\_TRACELEN : Length of trace array
- TSK\_STATUS is the task status. The return values are:
  - PG\_RUNACCEPT : Run request has been accepted

- PG\_ABORTING : Abort has been accepted
- PG\_RUNNING : Task is running
- PG\_PAUSED : Task is paused
- PG\_ABORTED : Task is aborted
- TSK\_PROGTYPE is the program type. The return values are:
  - PG\_NOT\_EXEC : Program has not been executed yet
  - PG\_MNEMONIC : Teach pendant program is or was executing
  - PG\_AR\_KAREL : KAREL program is or was executing
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** [Chapter 17, MULTI-TASKING](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.7.25 GET\_USEC\_SUB Built-In Procedure

---

**Purpose:** Returns an INTEGER value indicating the elapsed time in microseconds (1/1,000,000).

**Syntax:** us\_delta = GET\_USEC\_SUB(us2, us1)

Function Return Type: INTEGER

Input/Output Parameters:

[in] us2: INTEGER

[in] us1: INTEGER

%ENVIRONMENT Group: TIM

**Details:**

- us2 is the second time returned from GET\_USEC\_TIM.
- us1 is the first time returned from GET\_USEC\_TIM.
- The returned value is the INTEGER representation of the elapsed time us2 - us1 in microseconds.
- This is intended to measure fast operations. The result will wrap after 2 minutes and will no longer be valid.

**Example:** The following example measures the amount of time in microseconds to increment a number.

```
i = 0
us1 = GET_USEC_TIM
i = i + 1
us_delta = GET_USEC_SUB(GET_USEC_TIM, us1)
WRITE ('Time to increment a number: ', us_delta, ' us', CR)
```

**Figure A.7.25 GET\_USEC\_SUB Built-In Function**

## A.7.26 GET\_USEC\_TIM Built-In Function

---

**Purpose:** Returns an INTEGER value indicating the current time in microseconds (1/1,000,000) from within the KAREL system.

**Syntax:** us = GET\_USEC\_TIM

Function Return Type: I NTEGER

Input/Output Parameters:

None

%ENVIRONMENT Group : TIM

**Details:**

- The returned value is the INTEGER representation of the current time in microseconds stored in the KAREL system.
- This function is used with the GET\_USEC\_SUB built-in function to determine the elapsed time of an operation.

## **A.7.27 GET\_VAR Built-In Procedure**

**Purpose:** Allows a KAREL program to retrieve the value of a specified variable

**Syntax:** GET\_VAR(entry, prog\_name, var\_name, value, status)

Input/Output Parameters:

[in,out] entry : INTEGER

[in] prog\_name : STRING

[in] var\_name : STRING

[out] value : Any valid KAREL data type except PATH

[out] status : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- entry returns the entry number in the variable data table of var\_name in the device directory where var\_name is located. This variable should not be modified.
- prog\_name specifies the name of the program that contains the specified variable. If prog\_name is blank, it will default to the current task name being executed. Set the prog\_name to '\*SYSTEM\*' to get a system variable. prog\_name can also access a system variable on a robot in a ring.
- var\_name must refer to a static, program variable.
- var\_name can contain node numbers, field names, and/or subscripts.
- If both var\_name and value are ARRAYS, the number of elements copied will equal the size of the smaller of the two arrays.
- If both var\_name and value are STRINGS, the number of characters copied will equal the size of the smaller of the two strings.
- If both var\_name and value are STRUCTUREs of the same type, value will be an exact copy of var\_name.
- value is the value of var\_name.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- If the value of var\_name is uninitialized, then value will be set to uninitialized and status will be set to 12311.
- The designated names of all the robots can be found in the system variable \$PH\_MEMBERS[]/. This also include information about the state of the robot. The ring index is the array index for this system

variable. KAREL users can write general purpose programs by referring to the names and other information in this system variable rather than explicit names.

**See Also:** [Section A.19.38, SET\\_VAR Built-In Procedure](#), *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for information on accessing system variables on a robot in a ring.

**⚠ CAUTION**

Using GET\_VAR to modify system variables could cause unexpected results.

**Example 1:** To access \$TP\_DEFPROG on the MHROB03 robot in a ring, see [Figure A.7.27 \(a\)](#).

```
GET_VAR(entry, '\\MHROB03\*system*', '$TP_DEFPROG', strvar, status)
```

**Figure A.7.27 (a) Accessing \$TP\_DEFPROG on MHROB03**

**Example 2:** [Figure A.7.27 \(b\)](#) displays two programs, util\_prog and task. The program util\_prog uses a FOR loop to increment the value of the INTEGER variable num\_of\_parts. util\_prog also assigns values to the ARRAY part\_array. The program task uses two GET\_VAR statements to retrieve the values of num\_of\_parts and part\_array[3]. The value of num\_of\_parts is assigned to the INTEGER variable count and part\_array[3] is assigned to the STRING variable part\_name. The last GET\_VAR statement places the value of count into another INTEGER variable newcount.

```
PROGRAM util_prog
  VAR
    j, num_of_parts : INTEGER
    part_array      : ARRAY[5] OF STRING[10]
  BEGIN
    num_of_parts = 0

    FOR j = 1 to 20 DO
      num_of_parts = num_of_parts + 1
    ENDFOR
    part_array[1] = 10
    part_array[2] = 20
    part_array[3] = 30
    part_array[4] = 40
    part_array[5] = 50
  END util_prog
PROGRAM task
  VAR
    entry, status      : INTEGER
    count, new_count  : INTEGER
    part_name         : STRING[20]
  BEGIN
    GET_VAR(entry, 'util_prog', 'part_array[3]', part_name, status)
    WRITE('Part Name is Now....>', part_name, cr)

    GET_VAR(entry, 'util_prog', 'num_of_parts', count, status)
    WRITE('COUNT Now Equals....>', count, cr)

    GET_VAR(entry, 'task', 'count', new_count, status)

  END task
```

**Figure A.7.27 (b) GET\_VAR Built-In Procedure**

In [Figure A.7.27 \(c\)](#), an ARRAY [ipgetset]set\_data[x,y] is set on all robots in the ring from all robots in the ring. In this array, x is the source robot index and y is the destination robot index:

```

FOR idx = 1 TO $PH_ROSIP.$NUM_MEMBERS DO
    IF idx = $PH_ROSIP.$MY_INDEX THEN
        -- This will work but it this robot so is inefficient
    ELSE
        SELECT $PH_MEMBERS[idx].$STATE OF
        CASE (0) : -- Offline
            sstate = ' Offline'
        CASE (1) : -- Online
            sstate = ' Online '
        CASE (2) : -- Synchronized
            sstate = ' Synch '
            CNV_INT_STR(idx, 1, 10, sidx)
            prog_name = '\\\' + $PH_MEMBERS[idx].$NAME + '\\ipgetset'
            var_name = 'set_data[' + smy_index + ',' + sidx + ']'
            GET_VAR(entry, prog_name, var_name,
            set_data[$PH_ROSIP.$MY_INDEX,
                idx], status[idx])
            IF status[idx] = 0 THEN
                IF uninit(set_data[$PH_ROSIP.$MY_INDEX, idx]) THEN
                    set_data[$PH_ROSIP.$MY_INDEX, idx] = 0
                ELSE
                    set_data[$PH_ROSIP.$MY_INDEX, idx] =
                    set_data[$PH_ROSIP.$MY_INDEX,
                        idx] + 1
                ENDIF
                SET_VAR(entry, prog_name, var_name,
                set_data[$PH_ROSIP.$MY_INDEX, idx],
                status[idx])
            ENDIF
        ENDSELECT
    ENDIF
ENDFOR

```

**Figure A.7.27 (c) GET\_VAR SET\_VAR Built-In Procedure**

**Example 3:** GET\_VAR and SET\_VAR can also be used to set register values.

This will work for the local robot with the program names \*posreg\* and \*numreg\*. For the local robot this has similar functionality to the GET\_POS\_REG, GET\_REG and SET\_REG, SET\_POS\_REG built-ins. The built-ins only work for the local robot. You can access robots in the ring via GET\_VAR and SET\_VAR by using the robot name as part of the program name.

For the case of GET\_VAR on numeric registers, the type of the KAREL variable must match the type of the register or the error, Incompatible value is returned. In the example below if numeric register 10 is a real value an error will be returned and the real value will be set in the error case.

If a position register is locked and you attempt to set it, the error position register locked is returned. See [Figure A.7.27 \(d\)](#).

```

program GETREG
%nolockgroup
VAR
entry: integer
status: integer
int_data: integer
real_data: real

```

```

posext_data: xyzwprext
BEGIN
    GET_VAR(entry, '\\mhrob01\*numreg*', '$NUMREG[10]', int_data,
status)
    IF status <> 0 THEN
        GET_VAR(entry, '\\mhrob01\*numreg*', '$NUMREG[10]', real_data,
status)
    ENDIF
    GET_VAR(entry, '\\mhrob01\*posreg*', '$POSREG[1, 10]',
posext_data, status)
    SET_VAR(entry, '\\mhrob01\*numreg*', '$NUMREG[20]', int_data,
status)
    SET_VAR(entry, '\\mhrob01\*numreg*', '$NUMREG[21]', real_data,
status)
    SET_VAR(entry, '\\mhrob01\*posreg*', '$POSREG[1, 20]',
posext_data, status)
end GETREG

```

**Figure A.7.27 (d) Using GET\_VAR and SET\_VAR To Set Register Values**

## A.7.28 GO TO Statement

**Purpose:** Transfers control to a specified statement

**Syntax:** || GO TO | GOTO || stmmt\_label

where:

stmmt\_label : A valid KAREL identifier

**Details:**

- stmmt\_label must be defined in the same routine or program body as the GO TO statement.
- Label identifiers are followed by double colons (::). Executable statements may or may not follow on the same line.
- GOTO should only be used in special circumstances where normal control structures such as WHILE, REPEAT, and FOR loops would be awkward or difficult to implement.

**See Also:** [Section 2.1.5, Labels](#) , for more information on rules for labels, [Appendix E, SYNTAX DIAGRAMS](#) , for additional syntax information

**Example:** The following example moves the TCP from one position to another depending on the status of DIN[1].

```

BEGIN
    IF NOT DIN[1] THEN
        move_to_p1 — Call TP program to do move
    ELSE
        GO TO end_it
    ENDIF
    IF NOT DIN[1] THEN
        move_to_p2 — Call TP program to do move
    ELSE
        GO TO end_it

```

```
ENDIF  
END_IT::
```

Figure A.7.28 GO TO Statement

## A.8 - H - KAREL LANGUAGE DESCRIPTION

### A.8.1 HOLD Action

**Purpose:** Causes the current motion to be held and prevents subsequent motions from starting

**Syntax:** HOLD <GROUP[n,{n}]>

**Details:**

- Any motion in progress is held. Robot and auxiliary or extended axes decelerate to a stop.
- An attempted motion after a HOLD is executed is also held. HOLD cannot be overridden by a condition handler which issues a motion.
- HOLD is released using the UNHOLD statement or action.
- If the group clause is not present, all groups for which the task has control (when the condition is defined) will be canceled.
- If a motion that is held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are held.
- Motion cannot be held for a different task.

**See Also:** [Chapter 8, POSITION DATA](#), for more information on starting and stopping motions

**Example:** The following example shows a condition handler that holds motion when DIN[1] turns on.

```
CONDITION[1] :  
WHEN DIN[1] = ON DO  
    HOLD  
ENDCONDITION
```

Figure A.8.1 HOLD Action

### A.8.2 HOLD Statement

**Purpose:** Causes the current motion to be held and prevents subsequent motions from starting

**Syntax:** HOLD <GROUP[n,{n}]>

**Details:**

- Any motion in progress is held. Robot and auxiliary or extended axes decelerate to a stop.
- An attempted motion after a HOLD is executed is also held. HOLD cannot be overridden by a condition handler which issues a motion.
- HOLD is released using the UNHOLD statement or action.
- All held motions are canceled if a RELEASE statement is executed while motion is held.
- If the group clause is not present, all groups for which the task has control will be canceled.

- If a motion that is stopped, resumed, canceled, or held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are stopped, resumed, canceled, or held.
- Motion cannot be held for a different task.

**See Also:** [Chapter 8, POSITION DATA](#), for more information on starting and stopping motions, [Appendix E, SYNTAX DIAGRAMS](#) for additional syntax information

## A.9 - I - KAREL LANGUAGE DESCRIPTION

---

### A.9.1 IF ... ENDIF Statement

---

**Purpose:** Executes a sequence of statements if a BOOLEAN expression is TRUE; an alternate sequence can be executed if the condition is FALSE.

**Syntax:** IF bool\_exp THEN

```
{ true_stmnt } <ELSE
{ false_stmnt } >ENDIF
```

where:

bool\_exp : BOOLEAN

true\_stmnt : An executable KAREL statement

false\_stmnt : An executable KAREL statement

#### Details:

- If bool\_exp evaluates to TRUE, the statements contained in the true\_stmnt are executed. Execution then continues with the first statement after the ENDIF.
- If bool\_exp evaluates to FALSE and no ELSE clause is specified, execution skips directly to the first statement after the ENDIF.
- If bool\_exp evaluates to FALSE and an ELSE clause is specified, the statements contained in the false\_stmnt are executed. Execution then continues with the first statement after the ENDIF.
- IF statements can be nested in either true\_stmnt or false\_stmnt.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for additional syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING  
\(DOUT\\_EX.KL\)](#)

## A.9.2 IN Clause

---

**Purpose:** Specifies where a variable will be created

**Syntax:** `IN (CMOS | DRAM | SHADOW)`

**Details:**

- The `IN` clause can be part of a variable declaration. It should be specified before the `FROM` clause.
- `IN CMOS` specifies that the variable will be created in permanent memory.
- `IN DRAM` specifies that the variable will be created in temporary memory.
- `IN SHADOW` specifies that any changes made to the variable will be maintained in CMOS. Writes to this type of variable are slower but reads are much faster. This a good memory type to use for configuration parameters that are currently in CMOS.
- `IN UNINIT_DRAM` specifies that a DRAM variable is UNINITIALIZED at startup.
- If the `IN` clause is not specified all variables are created in temporary memory; unless the `%CMOSVARS` or `%SHADOWVARS` directive is specified, in which case all variables will be created in permanent memory.
- The `IN` clause cannot be used when declaring variables in the declaration section of a routine.

**See Also:** [Section 1.3.1, Memory](#), [Section A.3.18, %CMOSVARS Translator Directive](#), [Section A.19.39, %SHADOWVARS Translator Directive](#)

**Example:** Refer to the following sections for detailed program examples:

`IN DRAM`, [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

`IN CMOS`, [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#) or [Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

## A.9.3 %INCLUDE Translator Directive

---

**Purpose:** Inserts other files in a program at translation time.

**Syntax:** `%INCLUDE file_spec`

**Details:**

- `file_spec` is the name of the file to include. It has the following details:
  - The file name specified must be no longer than 36 characters.
  - The file type defaults to `.KL`, and so it does not appear in the directive.
- The `%INCLUDE` directive must appear on a line by itself.
- The specified files usually contain declarations, such as `CONST` or `VAR` declarations. However, they can contain any portion of a program including executable statements and even other `%INCLUDE` directives.
- Included files can themselves include other files up to a maximum depth of three nested included files. There is no limit on the total number of included files.

- When the KAREL language translator encounters a %INCLUDE directive during translation of a file, it begins translating the included file just as though it were part of the original file. When the entire file has been included, the translator resumes with the original file.
- Some examples in Appendix A reference the following include files:
  - %INCLUDE FR:\klevkmsk
  - %INCLUDE FR:\klevkeys
  - %INCLUDE FR:\klevccdf
  - %INCLUDE FR:\kliotyps

These files contain constants that can be used in your KAREL programs. If you are translating on the controller, you can include them directly from the FROM disk.

The include files are also installed with ROBOGUIDE into the support folder, and are copied to the hard disk as part of the installation process.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.9.4 INDEX Built-In Function

**Purpose:** Returns the index for the first character of the first occurrence of a specified STRING argument in another specified STRING argument. If the argument is not found, a 0 value is returned.

**Syntax:** INDEX(main, find)

Function Return Type: INTEGER

Input/Output Parameters:

[in] main : STRING

[in] find : STRING

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the index position in main corresponding to the first character of the first occurrence of find or 0 if find does not occur in main.

**Example:** The following example uses the INDEX built-in function to look for the first occurrence of the string 'Old' in part\_desc.

```
class_key = 'Old'
part_desc = 'Refurbished Old Part'
IF INDEX(part_desc, class_key) > 0 THEN
```

```
in_class = TRUE  
ENDIF
```

Figure A.9.4 INDEX Built-In Function

## A.9.5 INI\_DYN\_DISB Built-In Procedure

**Purpose:** Initiates the dynamic display of a BOOLEAN variable. This procedure displays elements of a STRING ARRAY depending of the current value of the BOOLEAN variable.

**Syntax:** INI\_DYN\_DISB (b\_var, window\_name, field\_width, attr\_mask, char\_size, row, col, interval, strings, status)

Input/Output Parameters:

[in] b\_var : BOOLEAN  
[in] window\_name : STRING  
[in] field\_width : INTEGER  
[in] attr\_mask : INTEGER  
[in] char\_size : INTEGER  
[in] row : INTEGER  
[in] col : INTEGER  
[in] interval : INTEGER  
[in] strings : ARRAY OF STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : UIF

**Details:**

- The dynamic display is initiated based on the value of b\_var:
  - If b\_var is FALSE, strings[1] is displayed.
  - If b\_var is TRUE, strings[2] is displayed.
  - If b\_var is uninitialized, a string of \*'s is displayed.
  - Both b\_var and strings must be static (not local) variables.
- window\_name must be a previously defined window name. See [Section 7.9.1, USER Menu on the Teach Pendant](#) and [Section 7.9.2, USER Menu on the CRT/KB](#) for predefined window names.
- If field\_width is non-zero, the display is extended with blanks if the element of strings[n] is shorter than this specified width. The area is cleared when the dynamic display is canceled.
- attr\_mask is a bit-wise mask that indicates character display attributes. This should be one of the following constants:
  - 0 : Normal
  - 1 : Bold (Supported only on the CRT)
  - 2 : Blink (Supported only on the CRT)
  - 4 : Underline
  - 8 : Reverse video

- To have multiple display attributes, use the OR operator to combine the constant attribute values together. For example, to have the text displayed as bold and underlined use 1 OR 4.
- char\_size specifies whether data is to be displayed in normal, double-wide, or double-high, double-wide sizes. This should be one of the following constants:
  - 0 : Normal
  - 1 : Double-wide (Supported only on the CRT)
  - 2 : Double-high, double-wide
- row and col specify the location in the window in which the data is to be displayed.
- interval indicates the minimum time interval, in milliseconds, between updates of the display. This must be greater than zero. The actual time might be greater since the task that formats the display runs at a low priority.
- strings[n] contains the text that will be displayed.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.20, CNC\\_DYN\\_DISB Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.6 INI\_DYN\_DISE Built-In Procedure

---

**Purpose:** Initiates the dynamic display of an INTEGER variable. This procedure displays elements of a STRING ARRAY depending of the current value of the INTEGER variable.

**Syntax:** `INI_DYN_DISE (e_var, window_name, field_width, attr_mask, char_size, row, col, interval, strings, status)`

**Input/Output Parameters:**

[in] e\_var : INTEGER  
 [in] window\_name : STRING  
 [in] field\_width : INTEGER  
 [in] attr\_mask : INTEGER  
 [in] char\_size : INTEGER  
 [in] row : INTEGER  
 [in] col : INTEGER  
 [in] interval : INTEGER  
 [in] strings : ARRAY OF STRING  
 [out] status : INTEGER

%ENVIRONMENT Group : UIF

### Details:

- The dynamic display is initiated based on the value of e\_var.
  - If e\_var has a value of n, strings[n+1] is displayed.
  - If e\_var has a negative value, or a value greater than or equal to the length of the array of strings, a string of ' ? 's is displayed.

- Both `e_var` and strings must be static (not local) variables.
- Refer to the [Section A.9.5,INI\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.3.21, CNC\\_DYN\\_DISE Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.7 INI\_DYN\_DISI Built-In Procedure

**Purpose:** Initiate the dynamic display of an INTEGER variable in a specified window.

**Syntax:** `INI_DYN_DISI(i_var, window_name, field_width, attr_mask, char_size, row, col, interval, buffer_size, format, status)`

Input/Output Parameters:

[in] `i_var` : INTEGER  
[in] `window_name` : STRING  
[in] `field_width` : INTEGER  
[in] `attr_mask` : INTEGER  
[in] `char_size` : INTEGER  
[in] `row` : INTEGER  
[in] `col` : INTEGER  
[in] `interval` : INTEGER  
[in] `buffer_size` : INTEGER  
[in] `format` : STRING  
[out] `status` : INTEGER

%ENVIRONMENT Group : UIF

### Details:

- `i_var` is the INTEGER whose dynamic display is to be initiated.
- If `field_width` is non-zero, the display is extended with blanks if `i_var` is shorter than this specified width. The area is cleared when the dynamic display is canceled.
- `buffer_size` is not implemented.
- `format` is used to print out the variable. This can be passed as a literal enclosed in single quotes. The format string begins with a % and ends with a conversion character. Between the % and the conversion character there can be, in order:
  - Flags (in any order), which modify the specification:
    - – : specifies left adjustment of this field.
    - + : specifies that the number will always be printed with a sign.
    - 0 : specifies padding a numeric field width with leading zeroes.
  - A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.

- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.

The format specifier must contain one of the conversion characters in [Table A.9.7](#).

**Table A.9.7 Conversion Characters**

Character	Argument Type; Printed As
d	INTEGER; decimal number
o	INTEGER; unsigned octal notation (without a leading zero).
x,X	INTEGER; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
u	INTEGER; unsigned decimal notation.
s	STRING; print characters from the string until end of string or the number of characters given by the precision.
f	REAL; decimal notation of the form [-]mmm.ddddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e,E	REAL; decimal notation of the form [-]m.ddddde+xx or [-]m.dddddE+xx, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g,G	REAL; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal pointer are not printed.
%	no argument is converted; print a %.

- Refer to the [Section A.9.5, INI\\_DYN\\_DISB Built-In Procedure](#) for a description of the other parameters listed above.

**See Also:** [Section A.3.22, CNC\\_DYN\\_DISI Built-In Procedure](#), [Section A.4.9, DEF\\_WINDOW Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.8 INI\_DYN\_DISP Built-In Procedure

**Purpose:** Initiates the dynamic display of a value of a port in a specified window, based on the port type and port number.

**Syntax:** `INI_DYN_DISP (port_type, port_no, window_name, field_width, attr_mask, char_size, row, col, interval, strings, status)`

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] window\_name : STRING

[in] field\_width : INTEGER  
[in] attr\_mask : INTEGER  
[in] char\_size : INTEGER  
[in] row : INTEGER  
[in] col : INTEGER  
[in] interval : INTEGER  
[in] strings : ARRAY OF STRING  
[out] status :INTEGER

**Details:**

- port\_type specifies the type of port to be displayed. Codes are defined in FROM: KLIOTYPS.KL.
  - If the port\_type is a BOOLEAN port (such as DIN), If the is FALSE, strings[1] is displayed; If variable is TRUE, strings[2] is displayed.
  - If the port\_type is an INTEGER port (such as GIN), if the value of the port is n, strings[n+1] will be displayed. If the value of the port is greater than or equal to the length of the array of strings, a string of ' ? ' s is displayed.
- port\_no specifies the port number to be displayed.
- Refer to the [INI\\_DYN\\_DISP](#) built-in procedure for a description of other parameters listed above.

**See Also:** [Section A.3.23, CNC\\_DYN\\_DISP Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.9 INI\_DYN\_DISR Built-In Procedure

**Purpose:** Initiates the dynamic display of a REAL variable in a specified window.

**Syntax:** `INI_DYN_DISR(r_var, window_name, field_width, attr_mask, char_size, row, col, interval, buffer_size, format, status)`

Input/Output Parameters:

[in] r\_var : REAL  
[in] window\_name : STRING  
[in] field\_width : INTEGER  
[in] attr\_mask : INTEGER  
[in] char\_size : INTEGER  
[in] row : INTEGER  
[in] col : INTEGER  
[in] interval : INTEGER  
[in] buffer\_size :INTEGER  
[in] format :STRING  
[out] status :INTEGER

%ENVIRONMENT Group: UIF

**Details:**

- r\_var is the REAL variable whose dynamic display is to be initiated.
- If field\_width is non-zero, the display is extended with blanks if r\_var is shorter than this specified width. The area is cleared when the dynamic display is canceled.
- Refer to the `INI_DYN_DISI` built-in procedure for a description of other parameters listed above.

**See Also:** [Section A.3.24, CNC\\_DYN\\_DISR Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.10 INI\_DYN\_DISS Built-In Procedure

---

**Purpose:** Initiates the dynamic display of a STRING variable in a specified window.

**Syntax:** `INI_DYN_DISS(s_var, window_name, field_width, attr_mask, char_size, row, col, interval, buffer_size, format, status)`

Input/Output Parameters:

[in] s\_var : STRING  
 [in] window\_name : STRING  
 [in] field\_width : INTEGER  
 [in] attr\_mask : INTEGER  
 [in] char\_size : INTEGER  
 [in] row : INTEGER  
 [in] col : INTEGER  
 [in] interval : INTEGER  
 [in] buffer\_size : INTEGER  
 [in] format : STRING  
 [out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- s\_var is the STRING variable whose dynamic display is to be initiated.
- If field\_width is non-zero, the display is extended with blanks if s\_var is shorter than this specified width. The area is cleared when the dynamic display is canceled.
- Refer to the `INI_DYN_DISI` built-in procedure for a description of other parameters listed above.

**See Also:** [Section A.3.25, CNC\\_DYN\\_DISS Built-In Procedure](#) , [Section A.9.7, INI\\_DYN\\_DISI Built-In Procedure](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.9.11 INIT\_QUEUE Built-In Procedure

**Purpose:** Sets a queue variable entry to have no entries in the queue

**Syntax:** INIT\_QUEUE(queue\_t)

Input/Output Parameters:

[out] queue\_t : QUEUE\_TYPE

%ENVIRONMENT Group : PBQMGR

**Details:**

- queue\_t is the queue to be initialized

**See Also:** [Section A.7.16, GET\\_QUEUE Built-In Procedure](#) , [Section A.13.2, MODIFY\\_QUEUE Built-In Procedure](#) , [Section A.17.2, QUEUE\\_TYPE Data Type](#) , [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

**Example:** The following example initializes a queue called job\_queue.

```
PROGRAM init_queue_x
%environment PBQMGR
VAR
    job_queue FROM globals: QUEUE_TYPE
BEGIN
    INIT_QUEUE(job_queue)
END init_queue_x
```

Figure A.9.11 INIT\_QUEUE Built-In Procedure

## A.9.12 INIT\_TBL Built-In Procedure

**Purpose:** Initializes a table on the teach pendant

**Syntax:** INIT\_TBL(dict\_name, ele\_number, num\_rows, num\_columns, col\_data, inact\_array, change\_array, value\_array, vptr\_array, table\_data, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] ele\_number : INTEGER

[in] num\_rows : INTEGER

[in] num\_columns : INTEGER

[in] col\_data : ARRAY OF COL\_DESC\_T

[in] inact\_array : ARRAY OF BOOLEAN

[in] change\_array : ARRAY OF ARRAY OF BOOLEAN

[in] value\_array : ARRAY OF STRING

[out] vptr\_array : ARRAY OF ARRAY OF INTEGER

[in,out] table\_data : XWORK\_T

[out] status : INTEGER

%ENVIRONMENT Group : UIF

#### Details:

- The INIT\_TBL and ACT\_TBL built-in routines should only be used instead of DISCTRL\_TBL if special processing needs to be done with each keystroke or if function key processing needs to be done without exiting the table menu.
- INIT\_TBL must be called before using the ACT\_TBL built-in. INIT\_TBL does not need to be called if using the DISCTRL\_TBL built-in.
- dict\_name is the four-character name of the dictionary containing the table header.
- ele\_number is the element number of the table header.
- num\_rows is the number of rows in the table.
- num\_columns is the number of columns in the table.
- col\_data is an array of column descriptor structures, one for each column in the table. It contains the following fields:
  - item\_type: Data type of values in this column. The following data type constants are defined:
    - TPX\_INT\_ITEM — INTEGER type
    - TPX\_REL\_ITEM — REAL type
    - TPX\_FKY\_ITEM — Function key enumeration type
    - TPX\_SUB\_ITEM — Subwindow enumeration type
    - TPX\_KST\_ITEM — KAREL STRING type
    - TPX\_KSL\_ITEM — KAREL STRING label type (can select, not edit)
    - TPX\_KBL\_ITEM — KAREL BOOLEAN type
    - TPX\_BYT\_ITEM — BYTE type
    - TPX\_SHT\_ITEM — SHORT type
    - TPX\_PBL\_ITEM — Port BOOLEAN type
    - TPX\_PIN\_ITEM — Port INTEGER type
  - start\_col: Starting character column (1..40) of the display field for this data column.
  - field\_width: Width of the display field for this data column.
  - num\_ele: Dictionary element used to display values for certain data types. The format of the dictionary elements for these data types are as follows:
    - TPX\_FKY\_ITEM: The enumerated values are placed on function key labels. There can be up to two pages of function key labels, for a maximum of 10 labels. Each label is a string of up to 8 characters. However, the last character of a label which is followed by another label should be left blank or the two labels will run together.
    - A single dictionary element defines all of the label values. Each value must be put on a separate line using &new\_line. The values are assigned to the function keys F1..F5, F6..F10 and the numeric values 1..10 in sequence. Unlabeled function keys should be left blank. If there are any labels on the second function key page, F6..F10, the labels for keys 5 and 10 must have the character > in column 8. If there are no labels on keys F6..F10, lines do not have to be specified for any key label after the last non-blank label.

```
$ example_fkey_label_c
"" &new_line
"F2" &new_line
"F3" &new_line
"F4" &new_line
"F5 >" &new_line
"F6" &new_line
```

```
"F7" &new_line
"" &new_line
"" &new_line
">"
```

**Figure A.9.12 (a) Example 1**

- TPX\_SUB\_ITEM: The enumerated values are selected from a subwindow on the display device. There can be up to 5 subwindow pages, for a maximum of 35 values. Each value is a string of up to 16 characters.
- A sequence of consecutive dictionary elements, starting with enum\_dict, define the values. Each value must be put in a separate element, and must not end with &new\_line. The character are assigned the numeric values 1..35 in sequence. The last dictionary element must be "\a".

```
$ example_sub_win_enum_c
"Red"
$
"Blue"
$
"Green"
$
"Yellow"
$
"\a"
```

**Figure A.9.12 (b) Example 2**

- TPX\_KBL\_ITEM, TPX\_PBL\_ITEM: The true and false values are placed on function key labels F4 and F5, in that order. Each label is a string of up to 8 characters. However, the last character of the true label should be left blank or the two labels will run together.
- A single dictionary element the label values. Each value must be put on a separate line using &new\_line, with the false value first.

```
$ example_boolean_c
"OFF" &new_line
"ON"
```

**Figure A.9.12 (c) Example 3**

- enum\_dict : Dictionary name used to display data types TPX\_FKY\_ITEM, TPX\_SUB\_ITEM, TPX\_KBL\_ITEM, or TPX\_PBL\_ITEM
- format\_spec: Format string is used to print out the data value. The format string contains a format specifier. The format string can also contain any desired characters before or after the format specifier. The format specifier itself begins with a % and ends with a conversion character. Between the % and the conversion character there may be, in order:
  - Flags (in any order), which modify the specification:
    - - : specifies left adjustment of this field.
    - + : specifies that the number will always be printed with a sign.
    - space: if the first character is not a sign, a space will be prefixed.
    - 0 : specifies padding a numeric field width with leading zeroes.
  - A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
  - A period, which separates the field width from the precision.

- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- The format specifier must contain one of the conversion characters in the following table:

**Table A.9.12 Conversion Characters**

Character	Argument Type; Printed As
d	INTEGER; decimal number.
o	INTEGER; unsigned octal notation (without a leading zero).
x,X	INTEGER; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
u	INTEGER; unsigned decimal notation.
s	STRING; print characters from the string until end of string or the number of characters given by the precision.
f	REAL; decimal notation of the form [-] mmm . ddddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e,E	REAL; decimal notation of the form [-] m . dddde+ -xx or [-] m . ddddE+ -xx, where the number of d's is given by the precision. The default precision is 6; a precision of 0
g,G	REAL; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal pointer are not printed.
%	no argument is converted; print a %.

"%d"

or

"%-10s"

**Figure A.9.12 (d) Example 4**

- The format specifiers which can be used with the data types specified in the item\_type field in col\_data are as follows:

- TPX\_INT\_ITEM %d, %o, %x, %X, %u
- TPX\_REL\_ITEM %f, %e, %E, %g, %G
- TPX\_FKY\_ITEM %s
- TPX\_SUB\_ITEM %s
- TPX\_KST\_ITEM %s
- TPX\_KSL\_ITEM %s
- TPX\_KBL\_ITEM %s
- TPX\_BYT\_ITEM %d, %o, %x, %X, %u, %c
- TPX\_SHT\_ITEM %d, %o, %x, %X, %u
- TPX\_PBL\_ITEM %s
- TPX\_PIN\_ITEM %d, %o, %x, %X, %u

- max\_integer: Maximum value if data type is TPX\_INT\_ITEM, TPX\_BYT\_ITEM, or TPX\_SHT\_ITEM.
- min\_integer: Minimum value if data type is TPX\_INT\_ITEM, TPX\_BYT\_ITEM, or TPX\_SHT\_ITEM.
- max\_real: Maximum value for reals.
- min\_real: Minimum value for reals.
- clear\_flag: If data type is TPX\_KST\_ITEM, 1 causes the field to be cleared before entering characters and 0 causes it not to be cleared.
- lower\_case: If data type is TPX\_KST\_ITEM, 1 allows the characters to be input to the string in upper or lower case and 0 restricts them to upper case.
- inact\_array is an array of booleans that corresponds to each column in the table.
  - You can set each boolean to TRUE which will make that column inactive. This means the column cannot be cursored to.
  - The array size can be less than or greater than the number of items in the table.
  - If inact\_array is not used, then an array size of 1 can be used, and the array does not need to be initialized.
- change\_array is a two dimensional array of booleans that corresponds to formatted data item in the table.
  - If the corresponding value is set, then the boolean will be set to TRUE, otherwise it is set to FALSE. You do not need to initialize the array.
  - The array size can be less than or greater than the number of data items in the table.
  - If change\_array is not used, then an array size of 1 can be used.
- value\_array is an array of variable names that correspond to the columns of data in the table. Each variable name can be specified as '[prog\_name]var\_name'.
  - [prog\_name] specifies the name of the program that contains the specified variable. If [prog\_name] is not specified, then the current program being executed is used.
  - var\_name must refer to a static, global program variable.
  - var\_name can contain node numbers, field names, and/or subscripts.
  - Each of these named variables must be a KAREL array of length num\_rows. Its data type and values should be consistent with the value of the item\_type field in col\_data for the corresponding column, as follows:
    - TPX\_INT\_ITEM: ARRAY OF INTEGER containing the desired values.
    - TPX\_REL\_ITEM: ARRAY OF REAL containing the desired values.
    - TPX\_FKY\_ITEM: ARRAY OF INTEGER with values referring to items in the dictionary element specified in the enum\_ele field in col\_data . There can be at most two function key pages, or 10 possible function key enumeration values.
    - TPX\_SUB\_ITEM: ARRAY OF INTEGER with values referring to items in the dictionary element specified in the enum\_ele field in col\_data . There can be at most 28 subwindow enumeration values.
    - TPX\_KST\_ITEM: ARRAY OF STRING containing the desired values.
    - TPX\_KSL\_ITEM: ARRAY OF STRING containing the desired values.
    - TPX\_KBL\_ITEM: ARRAY OF BOOLEAN containing the desired values. The dictionary element specified by the enum\_ele field in col\_data should have exactly two elements, with the false item first and the true item second.
    - TPX\_BYT\_ITEM: ARRAY OF BYTE containing the desired values.
    - TPX\_SHT\_ITEM: ARRAY OF SHORT containing the desired values.

- TPX\_PBL\_ITEM: ARRAY OF STRING containing the names of the ports, for example DIN[5]
- TPX\_PIN\_ITEM: ARRAY OF STRING containing the names of the ports, for example GOUT[3].
- TPX\_BYT\_ITEM: ARRAY OF BYTE containing the desired values.
- TPX\_SHT\_ITEM: ARRAY OF SHORT containing the desired values.
- TPX\_PBL\_ITEM: ARRAY OF STRING containing the names of the ports, for example DIN[5].
- TPX\_PIN\_ITEM: ARRAY OF STRING containing the names of the ports, for example GOUT[3].
- vptr\_array is an array of integers that corresponds to each variable name in value\_array. **Do not change this data; it is used internally.**
- table\_data is used to display and control the table. **Do not change this data; it is used internally.**
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** In this example, TPXTABEG.TX is loaded as 'XTAB' on the controller. TPEXTBL calls INIT\_TBL to initialize a table with five columns and four rows. It calls ACT\_TBL in a loop to read and process each key pressed.

```
-----
TPXTABEG.TX
-----
$title
&reverse "DATA Test Schedule" &standard &new_line
"E1: " &new_line
"      W(mm) TEST C(%%) G(123456) COLOR"
^1
?2
$function_keys
"f1"      &new_line
"f2"      &new_line
"f3"      &new_line
"f4"      &new_line
" HELP >" &new_line
"f6"      &new_line
"f7"      &new_line
"f8"      &new_line
"f9"      &new_line
"f10     >"
$help_text "Help text goes here...
$enum1
"" &new_line
"" &new_line
"TRUE" &new_line
"FALSE" &new_line
"""
$enum2
"Red"
$
"Blue"
$
"Green"
$
"Yellow"
$
"Brown"
$
```

```

"Pink"
$
"Mauve"
$
"Black"
$
"....."
-----
TPEXTBL.KL
-----
PROGRAM tpextbl
%ENVIRONMENT uif
%INCLUDE FROM:\klevccdf
%INCLUDE FROM:\klevkeysVAR
    dict_name: STRING[6]
    ele_number: INTEGER
    num_rows: INTEGER
    num_columns: INTEGER
    col_data: ARRAY[5] OF COL_DESC_T
    inact_array: ARRAY[5] OF BOOLEAN
    change_array: ARRAY[4,5] OF BOOLEAN
    value_array: ARRAY[5] OF STRING[26]
    vptr_array: ARRAY[4,5] OF INTEGER
    table_data: XWORK_T
    status: INTEGER
    action: INTEGER
    def_item: INTEGER
    term_char: INTEGER
    attach_sw: BOOLEAN
    save_action: INTEGER
    done: BOOLEAN
    value1: ARRAY[4] OF INTEGER
    value2: ARRAY[4] OF INTEGER
    value3: ARRAY[4] OF REAL
    value4: ARRAY[4] OF STRING[10]
    value5: ARRAY[4] OF INTEGER
BEGIN
    def_item = 1
    value_array[1] = 'value1'
    value_array[2] = 'value2'
    value_array[3] = 'value3'
    value_array[4] = 'value4'
    value_array[5] = 'value5'
    value1[1] = 21
    value1[2] = 16
    value1[3] = 1
    value1[4] = 4
    value2[1] = 3
    value2[2] = 2
    value2[3] = 3
    value2[4] = 2
    value3[1] = -13
    value3[2] = 4.1
    value3[3] = 23.9
    value3[4] = -41
    value4[1] = 'XXX---'
    value4[2] = '--X-X-'
    value4[3] = 'XXX-XX'
    value4[4] = '-X-X--'
    value5[1] = 1
    value5[2] = 1

```

```

value5[3] = 2
value5[4] = 3
inact_array[1] = FALSE
inact_array[2] = FALSE
inact_array[3] = FALSE
inact_array[4] = FALSE
inact_array[5] = FALSE
col_data[1].item_type = TPX_INT_ITEM
col_data[1].start_col = 6
col_data[1].field_width = 4
col_data[1].format_spec = '%3d'
col_data[1].max_integer = 99
col_data[1].min_integer = -99
col_data[2].item_type = TPX_FKY_ITEM
col_data[2].start_col = 12
col_data[2].field_width = 5
col_data[2].format_spec = '%s'
col_data[2].enum_ele = 3 -- enum1 element number
col_data[2].enum_dict = 'XTAB'
col_data[3].item_type = TPX_REL_ITEM
col_data[3].start_col = 18
col_data[3].field_width = 5
col_data[3].format_spec = '%3.1f'
col_data[4].item_type = TPX_KST_ITEM
col_data[4].start_col = 26
col_data[4].field_width = 6
col_data[4].format_spec = '%s'
col_data[5].item_type = TPX_SUB_ITEM
col_data[5].start_col = 34
col_data[5].field_width = 6
col_data[5].format_spec = '%s'
col_data[5].enum_ele = 4 -- enum2 element number
col_data[5].enum_dict = 'XTAB'
dict_name = 'XTAB'
ele_number = 0 -- title element number
num_rows = 4
num_columns = 5
def_item = 1
attach_sw = TRUE
INIT_TBL(dict_name, ele_number, num_rows, num_columns, col_data,
         inact_array, change_array, value_array, vptr_array,
         table_data, status)
IF status <> 0 THEN
    WRITE('INIT_TBL status = ', status, CR);
ELSE
    def_item = 1
    -- Initial display of table
    ACT_TBL(ky_disp_upd, def_item, table_data, term_char,
            attach_sw, status)
    IF status <> 0 THEN
        WRITE(CR, 'ACT_TBL status = ', status)
    ENDIF
ENDIF
IF status = 0 THEN
    -- Loop until a termination key is selected.
    done = FALSE
    action = ky_reissue -- read new key
    WHILE NOT done DO
        -- Read new key, act on it, and return it
        ACT_TBL(action, def_item, table_data, term_char,
                attach_sw, status) save_action = action

```

```

action = ky_reissue -- read new key
-- Debug only
WRITE TPERROR (CHR(cc_home) + CHR(cc_clear_win))
-- Process termination keys.
SELECT (term_char) OF
  CASE (ky_select, ky_new_menu):
    done = TRUE;
  CASE (ky_f1):
    -- Perform F1
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F1 pressed')
  CASE (ky_f2):
    -- Perform F2
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F2 pressed')
  CASE (ky_f3):
    -- Perform F3
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F3 pressed')
  CASE (ky_f4):
    -- Perform F4
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F4 pressed')
  CASE (ky_f5):
    -- Perform F5 Help
    action = ky_help
  CASE (ky_f6):
    -- Perform F6
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F6 pressed')
  CASE (ky_f7):
    -- Perform F7
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F7 pressed')
  CASE (ky_f8):
    -- Perform F8
    SET_CURSOR(TPERROR, 1, 1, status)
    WRITE TPERROR ('F8 pressed')
CASE (ky_f9):
  -- Perform F9
  SET_CURSOR(TPERROR, 1, 1, status)
  WRITE TPERROR ('F9 pressed')
CASE (ky_f10):
  -- Perform F10
  SET_CURSOR(TPERROR, 1, 1, status)
  WRITE TPERROR ('F10 pressed')
CASE (ky_undef):
  -- Process special keys.
  SELECT (save_action) OF
    CASE (ky_f1_s):
      -- Perform Shift F1
      SET_CURSOR(TPERROR, 1, 1, status)
      WRITE TPERROR ('F1 shifted pressed')
    ELSE:
      ENDSELECT
  ELSE:
    action = term_char -- act on this key
  ENDSELECT
ENDWHILE
IF term_char <> ky_new_menu THEN
-- Cancel the dynamic display

```

```

        ACT_TBL(ky_cancel, def_item, table_data, term_char,
                 attach_sw, status)
    ENDIF
ENDIF
END tpextbl

```

**Figure A.9.12 (e) INIT\_TBL Built-In Procedure**

## A.9.13 IN\_RANGE Built-In Function

**Purpose:** Returns a BOOLEAN value indicating whether or not the specified position argument can be reached by a group of axes

**Syntax:** IN\_RANGE(posn)

Function Return Type : BOOLEAN

Input/Output Parameters:

[in] posn : XYZWPREXT

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is TRUE if posn is within the work envelope of the group of axes; otherwise, FALSE is returned.
- The current \$UFRAME and \$UTOOL are applied to posn.

**See Also:** [Section A.3.7, CHECK\\_EPOS Built-In Procedure](#)

**Example:** The following example checks to see if the new position is in the work envelope before moving the TCP to it.

```

IF IN_RANGE(part_slot) THEN
    SET_POS_REG(1, part_slot, status)
    move_to_pr — Call TP program to move to PR[1]
ELSE WRITE('I can't get there!',CR)
ENDIF

```

**Figure A.9.13 IN\_RANGE Built-In Function**

## A.9.14 INSERT\_NODE Built-In Procedure

**Purpose:** Inserts an uninitialized node in the specified PATH argument preceding the specified path node number

**Syntax:** INSERT\_NODE(path\_var, node\_num, status)

Input/Output Parameters:

[in] path\_var : PATH

[in] node\_num : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PATHOP

**Details:**

- node\_num specifies the index number of the path node before which the new uninitialized node is to be inserted.
- The new node can be assigned values by directly referencing its NODEDATA structure.
- All nodes following the inserted node are renumbered.
- Valid node\_num values are in the range node\_num => 1 and node\_num <= PATH\_LEN(path\_var).
- If node\_num is not a valid node number, status is returned with an error.
- If the program does not have enough RAM for an INSERT\_NODE request, an error will occur.
- If the program is paused, the INSERT\_NODE request is not retried.

**See Also:** [Section A.4.13, DELETE\\_NODE Built-In Procedure](#) , [Section A.1.15, APPEND\\_NODE Built-In Procedure](#)

**Example:** In the following example, the PATH\_LEN built-in is used to set the variable length equal to the number of nodes in path\_var. INSERT\_NODE inserts a new path node before the last node in path\_var.

```
length = PATH_LEN(path_var)
INSERT_NODE(path_var, length, status)
```

**Figure A.9.14 INSERT\_NODE Built-In Procedure**

## A.9.15 INSERT\_QUEUE Built-In Procedure

**Purpose:** Inserts an entry into a queue if the queue is not full

**Syntax:** INSERT\_QUEUE(value, sequence\_no, queue\_t, queue\_data, status)

Input/Output Parameters:

[in] value : INTEGER

[in] sequence\_no : INTEGER

[in,out] queue\_t : QUEUE\_TYPE

[in,out] queue\_data : ARRAY OF INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBQMGR

**Details:**

- value specifies the value to be inserted into the queue, queue\_t.
- sequence\_no specifies the sequence number of the entry before which the new entry is to be inserted.
- queue\_t specifies the queue variable for the queue.
- queue\_data specifies the array used to hold the data in the queue. The length of this array determines the maximum number of entries in the queue.
- status is returned with **61002**, Queue is full, if there is no room for the entry in the queue, with **61003**, Bad sequence no, if the specified sequence number is not in the queue.

**See Also:** [Section A.13.2, MODIFY\\_QUEUE Built-In Procedure](#) , [Section A.1.16, APPEND\\_QUEUE Built-In Procedure](#) , [Section A.4.14, DELETE\\_QUEUE Built-In Procedure](#) , [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

**Example:** In the following example, the routine `ins_in_queue` adds an entry (value) to a queue (`queue_t` and `queue_data`) following the specified entry (`sequence_no`); it returns TRUE if this was successful; otherwise it returns FALSE.

```

PROGRAM ins_queue_x
%environment PBQMGR
ROUTINE ins_in_queue(value: INTEGER;
                      sequence_no: INTEGER;
                      queue_t: QUEUE_TYPE;
                      queue_data: ARRAY OF INTEGER): BOOLEAN
VAR
    status: INTEGER
BEGIN
    INSERT_QUEUE(value, sequence_no, queue_t, queue_data, status)
    return (status = 0)
END ins_in_queue
BEGIN
END ins_queue_x

```

**Figure A.9.15 INSERT\_QUEUE Built-In Procedure**

## A.9.16 INTEGER Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as INTEGER data type

**Syntax:** INTEGER

**Details:**

- An INTEGER variable or parameter can assume whole number values in the range -2147483648 through +2147483648.
- INTEGER literals consist of a series of digits, optionally preceded by a plus or minus sign. They cannot contain decimal points, commas, spaces, dollar signs (\$), or other punctuation characters. (See Table A.9.16 )

**Table A.9.16 Valid and Invalid INTEGER Literals**

Valid	Invalid	Reason
1	1.5	Decimal point not allowed (must be a whole number)
-2500450	-2,500,450	Commas not allowed
+65	+6 5	Spaces not allowed

- If an INTEGER argument is passed to a routine where a REAL parameter is expected, it is treated as a REAL and passed by value.
- Only INTEGER expressions can be assigned to INTEGER variables, returned from INTEGER function routines, or passed as arguments to INTEGER parameters.
- Valid INTEGER operators are:
  - Arithmetic operators (+, -, \*, /, DIV, MOD)
  - Relational operators (>, >=, =, <>, <, <=)
  - Bitwise operations (AND, OR, NOT)

**See Also:** [Chapter 5, ROUTINES](#) , for more information on passing by value, [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#) , for more information on format specifiers

**Example:** Refer to [Appendix B, KAREL EXAMPLE PROGRAMS](#) for detailed program examples.

## A.9.17 INV Built-In Function

**Purpose:** Used in coordinate frame transformations with the relative position operator (:) to determine the coordinate values of a POSITION in a frame that differs from the frame in which that POSITION was recorded

**Syntax:** INV(pos)

Function Return Type : POSITION

Input/Output Parameters:

[in] pos : POSITION

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the inverse of the pos argument.
- The configuration of the returned POSITION will be that of the pos argument.

**Example:** The following example uses the INV built-in to determine the POSITION of part\_pos with reference to the coordinate frame that has rack\_pos as its origin. Both part\_pos and rack\_pos were originally taught and recorded in User Frame. The robot is then instructed to move to that position.

```
PROGRAM p_inv
VAR
    rack_pos, part_pos, p1 : POSITION
BEGIN
    p1 = INV(rack_pos):part_pos
    SET_POS_REG(1, p1, status)
    move_to_pr1 -- Call TP program to move to PR[1]
END p_inv
```

Figure A.9.17 INV Built-In Function

## A.9.18 IO\_MOD\_TYPE Built-In Procedure

**Purpose:** Allows a KAREL program to determine the type of module in a specified rack/slot

**Syntax:** IO\_MOD\_TYPE(rack\_no, slot\_no, mod\_type, status)

Input/Output Parameters:

[in] rack\_no : INTEGER

[in] slot\_no : INTEGER

[out] mod\_type : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- rack\_no is the rack containing the port module. For process I/O boards, this is zero; for Allen-Bradley and Genius ports, this is 16.
- slot\_no is the slot containing the port module. For process I/O boards, this is the position of the board in the SLC-2 chain.
- mod\_type is the module type:
  - 6 A16B-2202-470
  - 7 A16B-2202-472
  - 8 A16B-2202-480
- status is returned with zero if the parameters are valid and there is a module or board with the specified rack/slot number.

**Example:** The following example returns to the caller the module in the specified rack and slot number.

```

PROGRAM iomodtype
%ENVIRONMENT IOSETUP
ROUTINE get_mod_type(rack_no: INTEGER;
                      slot_no: INTEGER;
                      mod_type: INTEGER) : INTEGER
VAR
    status: INTEGER
BEGIN
    IO_MOD_TYPE(rack_no, slot_no, mod_type, status)
    RETURN (status)
END get_mod_type
BEGIN
END iomodtype

```

**Figure A.9.18 IO\_MOD\_TYPE Built-In Procedure**

## A.9.19 IO\_STATUS Built-In Function

**Purpose:** Returns an INTEGER value indicating the success or type of failure of the last operation on the file argument

**Syntax:** IO\_STATUS(file\_id)

Function Return Type : INTEGER

Input/Output Parameters:

[in] file\_id : FILE

%ENVIRONMENT Group : PBCORE

**Details:**

- IO\_STATUS can be used after an OPEN FILE, READ, WRITE, CANCEL FILE, or CLOSE FILE statement. Depending on the results of the operation, it will return 0 if successful or one of the errors listed in the *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)*. Some of the common errors are shown in [Table A.9.19](#).

**Table A.9.19 IO\_STATUS Errors**

0	Last operation on specified file was successful
2021	End of file for RAM disk device

10006	End of file for floppy device
12311	Uninitialized variable
12324	Illegal open mode string
12325	Illegal file string
12326	File var is already used
12327	Open file failed
12328	File is not opened
12329	Cannot write the variable
12330	Write file failed
12331	Cannot read the variable
12332	Read data is too short
12333	Illegal ASCII string for read
12334	Read file failed
12335	Cannot open pre_defined file
12336	Cannot close pre_defined file
12338	Close file failed
12347	Read I/O value failed
12348	Write I/O value failed
12358	Timeout at read request
12359	Read request is nested
12367	Bad base in format

- Use READ file\_id(cr) to clear any IO\_STATUS error.
- If file\_id does not correspond to an opened file or one of the pre-defined files opened to the respective CRT/KB, teach pendant, and vision windows, the program is aborted with an error.

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLIST\\_EX.KL\)](#), for a detailed program example.

## A.10 - J - KAREL LANGUAGE DESCRIPTION

---

### A.10.1 J\_IN\_RANGE Built-In Function

---

**Purpose:** Returns a BOOLEAN value indicating whether or not the specified joint position argument can be reached by a group of axes

**Syntax:** J\_IN\_RANGE(posn)

Function Return Type : BOOLEAN

Input/Output Parameters:

[in] posn : JOINTPOS

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is TRUE if posn is within the work envelope; otherwise, FALSE is returned.

**See Also:** [Section A.9.13, IN\\_RANGE Built-In Function](#), [Section A.3.7, CHECK\\_EPOS Built-In Procedure](#)

## A.10.2 JOINTPOS Data Type

---

**Purpose:** Defines a variable, function return type, or routine parameter as JOINTPOS data type.

**Syntax:** JOINTPOS<n> <IN GROUP[m]>

**Details:**

- A JOINTPOS consists of a REAL representation of the position of each axis of the group, expressed in degrees or millimeters (mm).
- n specifies the number of axes, with 9 as the default. The size in bytes is  $8 + 4 * n$ .
- A JOINTPOS may be followed by IN GROUP[m], where m indicates the motion group with which the data is to be used. The default is the group specified by the %DEFGROUP directive or 1.
- CNV\_REL\_JPOS and CNV\_JPOS\_REL built-ins can be used to access the real values.
- A JOINTPOS can be assigned to other positional types. Note that some motion groups, for example single axis positioners, have no XYZWPR representation. If you attempt to assign a JOINTPOS to a XYZWPR or POSITION type for such a group, a run-time error will result.

**Example:** Refer to the following sections for detailed program examples:

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.10.3 JOINT2POS Built-In Function

---

**Purpose:** This routine is used to convert joint angles (in\_jnt) to a Cartesian position (out\_pos) by calling the forward kinematics routine.

**Syntax:** JOINT2POS (in\_jnt - Joint angles can be converted to Cartesian, uframe, utool, config\_ref, out\_pos, wjnt\_cfg, ext\_ang, and status).

Input/Output Parameters:

[in] in\_jnt : JOINTPOS

[in] uframe : POSITION

[in] utool : POSITION

[in] config\_ref : INTEGER

[out] out\_pos : POSITION

[out] wjnt\_cfg : CONFIG

[out] ext\_ang : ARRAY OF REAL

[out] status : INTEGER

%ENVIRONMENT Group : MOTN

**Details:**

- The input in\_jnt is defined as the joint angles to be converted to the Cartesian position.
- The input uframe is the user frame for the Cartesian position.
- The input utool is defined as the corresponding tool frame.
- The input config\_ref is an INTEGER representing the type of solution desired. The values listed below are valid. Also, the pre-defined constants in the parentheses can be used and the values can be added as required. One example includes: config\_ref=HALF\_SOLN+CONFIG\_TCP.
  - 0 : (FULL\_SOLN) = Default
  - 1 : (HALF\_SOLN) = Wrist joint (xyz456). This value does not calculate/use wpr.
  - 2 : (CONFIG\_TCP) = The Wrist Joint Config (up/down) is based on the fixed wrist.
  - 4 : (APPROX\_SOLN) = Approximate solution. This value reduce calculation time for some robots.
  - 8 : (NO\_TURNS) = Ignore wrist turn numbers. Use the closest path for joints 4, 5 and 6 (uses ref\_jnt).
  - 16 : (NO\_M\_TURNS) = Ignore major axis (J1 only) turn number. Use the closest path.
- The output out\_pos is the Cartesian position corresponding to the input joint angles.
- The output wjnt\_cfg is the wrist joint configuration. The value will be output when config\_ref corresponds to HALF\_SOLN.
- The output ext\_ang contains the values of the joint angles for the extended axes if they exist.
- The output status explains the status of the attempted operation. If it is not equal to 0, then an error has occurred.

## A.11 - K - KAREL LANGUAGE DESCRIPTION

### A.11.1 KCL Built-In Procedure

**Purpose:** Sends the KCL command specified by the STRING argument to KCL for execution.

**Syntax:** KCL (command, status)

Input/Output Parameters:

[in] command : STRING

[out] status : INTEGER

%ENVIRONMENT Group : KCLOP

**Details:**

- command must contain a valid KCL command.
- command cannot exceed 126 characters.
- Program execution waits until execution of the KCL command is completed or until an error is detected.
- All KCL commands are performed as if they were entered at the command level, with the exception of destructive commands, such as CLEAR ALL, for which no confirmation is required.

- status indicates whether the command was executed successfully.
- If a KCL command file is being executed and `$STOP_ON_ERR` is FALSE, the KCL built-in will continue to run to completion. The first error detected will be returned or a 0 if no errors occurred.

**See Also:** [Section A.11.2, KCL\\_NO\\_WAIT Built-In Procedure](#) , [Section A.11.3, KCL\\_STATUS Built-In Procedure](#)

**Example:** The following example will show programs and wait until finished. Status will be the outcome of this operation.

```
PROGRAM kcl_test
VAR
    command :STRING[20]
    status :INTEGER
BEGIN
    command = 'SHOW PROGRAMS'
    KCL(command, status)
END kcl_test
```

**Figure A.11.1 KCL Built-In Procedure**

Refer to [Figure 11.2.19 \(h\)](#) for another example.

## A.11.2 KCL\_NO\_WAIT Built-In Procedure

**Purpose:** Sends the KCL command specified by the STRING argument to KCL for execution, but does not wait for completion of the command before continuing program execution.

**Syntax:** `KCL_NO_WAIT (command, status)`

Input/Output Parameters:

[in] command : STRING  
 [out] status : INTEGER  
 %ENVIRONMENT Group : KCLOP

**Details:**

- command must contain a valid KCL command.
- status indicates whether KCL accepted the command.
- Program execution waits until KCL accepts the command or an error is detected.

**See Also:** [Section A.11.1, KCL Built-In Procedure](#) , [Section A.11.3, KCL\\_STATUS Built-In Procedure](#)

**Example:** The following example will load a program, but will not wait for the program to be loaded before returning. Status will indicate if the command was accepted or not.

```
PROGRAM kcl_test
VAR
    command :STRING[20]
    status :INTEGER
BEGIN
    command = 'Load prog test_1'
    KCL_NO_WAIT (command, status)
    delay 5000
```

```
status = KCL_STATUS
END kcl_test
```

Figure A.11.2 KCL\_NO\_WAIT Built-In Procedure

## A.11.3 KCL\_STATUS Built-In Procedure

**Purpose:** Returns the status of the last executed command from either KCL or KCL\_NO\_WAIT built-in procedures.

**Syntax:** KCL\_STATUS

Function Return Type: INTEGER

%ENVIRONMENT Group : KCLOP

**Details:**

- Returns the status of the last executed command from the KCL or KCL\_NO\_WAIT built-ins.
- Program execution waits until KCL can return the status.

**See Also:** [Section A.11.2, KCL\\_NO\\_WAIT Built-In Procedure](#) , [Appendix C, KCL COMMAND ALPHABETICAL DESCRIPTION](#)

## A.12 - L - KAREL LANGUAGE DESCRIPTION

### A.12.1 LN Built-In Function

**Purpose:** Returns the natural logarithm of a specified REAL argument

**Syntax:** LN(x)

Function Return Type: REAL

Input/Output Parameters:

[in] x : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the natural logarithm of x.
- x must be greater than zero. Otherwise, the program will be aborted with an error.

**Example:** The following example returns the natural logarithm of the input variable *a* and assigns it to the variable *b*.

```
WRITE(CR, CR, 'enter a number =')
READ(a,CR)
b = LN(a)
```

**Figure A.12.1 LN Built-In Function**

## A.12.2 LOAD Built-In Procedure

**Purpose:** Loads the specified file

**Syntax:** LOAD (file\_spec, option\_sw, status)

Input/Output Parameters:

[in] file\_spec : STRING

[in] option\_sw : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- file\_spec specifies the device, name, and type of the file to load. The following types are valid:
  - .TP – Teach pendant program
  - .PC – KAREL program
  - .VR – KAREL variables
  - .SV – KAREL system variables
  - .IO – I/O configuration data
  - no ext – KAREL program and variables
- option\_sw specifies the type of options to be done during loading.

The following value is valid for .TP files:

- 1 – If the program already exists, then it overwrites the program.
- If option\_sw is not 1 and the program exists, an error will be returned.

The following value is valid for .SV files:

- 1 – Converts system variables.

- option\_sw is ignored for all other types.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**NOTE**

For R-30iB Plus, R-30iB, and R-30iB Mate controllers, the KAREL option must be installed on the robot controller in order to load KAREL programs.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

## **A.12.3 LOAD\_STATUS Built-In Procedure**

---

**Purpose:** Determines whether the specified KAREL program and its variables are loaded into memory

**Syntax:** LOAD\_STATUS(prog\_name, loaded, initialized)

Input/Output Parameters:

[in] prog\_name : STRING

[out] loaded : BOOLEAN

[out] initialized : BOOLEAN

%ENVIRONMENT Group : PBCORE

**Details:**

- prog\_name must be a program and cannot be a routine.
- loaded returns a value of TRUE if prog\_name is currently loaded into memory. FALSE is returned if prog\_name is not loaded.
- initialized returns a value of TRUE if any variable within prog\_name has been initialized. FALSE is returned if all variables within prog\_name are uninitialized.
- If either loaded or initialized is FALSE, use the LOAD built-in procedure to load prog\_name and its variables.

**Example:** Refer to the following sections for detailed program examples:

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

## **A.12.4 LOCK\_GROUP Built-In Procedure**

---

**Purpose:** Locks motion control for the specified group of axes

**Syntax:** LOCK\_GROUP(group\_mask, status)

Input/Output Parameters:

[in] group\_mask : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- group\_mask specifies the group of axes to lock for the running task. The group numbers must be in the range of 1 to the total number of groups defined on the controller.
- The group\_mask is specified by setting the bit(s) for the desired group(s).

**Table A.12.4 Group\_mask Setting**

GROUP	DECIMAL	BIT
Group 1	1	1
Group 2	2	2
Group 3	4	3

To specify multiple groups select the decimal values, shown in [Table A.12.4](#), which correspond to the desired groups. Then connect them together using the OR operator. For example to specify groups 1, 3, enter "1 OR 4".

- Motion control is gained for the specified motion groups.
- If one or more of the groups cannot be locked, then an error is returned, and any available groups will be locked.
- Moving a group automatically locks the group if it has not been previously locked by another task.
- If a task tries to move a group that is already locked by another task, it will be paused.
- status explains the status of the attempted operation. If not equal to 0, an error occurred.

**Example:** The following example unlocks group 1, 2, and 3, and then locks group 3. Refer to [Chapter 17, MULTI-TASKING](#), for more examples.

```
%ENVIRONMENT MOTN
%ENVIRONMENT MULTI
VAR
    status: INTEGER
BEGIN
    REPEAT
        -- Unlock groups 1, 2, and 3
        UNLOCK_GROUP(1 OR 2 OR 4, status)
        IF status = 17040 THEN
            CNCL_STP_MTN -- or RESUME
        ENDIF
        DELAY 500
    UNTIL status = 0
    -- Lock only group 3
    LOCK_GROUP(4, status)
END lock_grp_ex
```

**Figure A.12.4 LOCK\_GROUP Built-In Procedure**

## A.12.5 %LOCKGROUP Translator Directive

**Purpose:** Specifies the motion group(s) to be locked when calling this program or a routine from this program.

**Syntax:** %LOCKGROUP = n, n ,...

**Details:**

- n is the number of the motion group to be locked.
- The range of n is 1 to the number of groups on the controller.
- When the program or routine is called, the task will attempt to get motion control for all the specified groups if it does not have them locked already. The task will pause if it cannot get motion control.
- If %LOCKGROUP is not specified, all groups will be locked.

- The %NOLOCKGROUP directive can be specified if no groups should be locked.

**See Also:** [Section A.14.5, %NOLOCKGROUP Translator Directive](#) , [Section A.12.4, LOCK\\_GROUP Built-In Procedure](#) , [Section A.21.5, UNLOCK\\_GROUP Built-In Procedure](#)

## A.13 - M - KAREL LANGUAGE DESCRIPTION

### A.13.1 MIRROR Built-In Function

**Purpose:** Determines the mirror image of a specified position variable.

**Syntax:** MIRROR (old\_pos, mirror\_frame, orientation\_flag)

Function Return Type: XYZWPREXT

Input/Output Parameters:

[in] old\_pos : POSITION

[in] mirror\_frame : POSITION

[in] orient\_flag : BOOLEAN

%ENVIRONMENT Group : MIR

**Details:**

- old\_pos and mirror\_frame must both be defined relative to the same user frame.
- old\_pos specifies the value whose mirror image is to be generated.
- mirror\_frame specifies the value across whose xz\_plane the image is to be generated.
- If orient\_flag is TRUE, both the orientation and location component of old\_pos will be mirrored. If FALSE, only the location is mirrored and the orientation of the new mirror-image position is the same as that of old\_pos.
- The returned mirrored position is not guaranteed to be a reachable position, since the mirrored position can be outside of the robot's work envelope.

**See Also:** Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function) (B-83284EN)*

**Example:** The following example gets the current position of the robot, creates a mirror frame, and generates a mirrored position which is mirrored about the y axis.

```
PROGRAM mir_exam
VAR
    cur_pos:    XYZWPREXT
    org_pos:    POSITION
    mir_pos:    XYZWPREXT
    mir_posa:   POSITION
    pos_frame:  XYZWPREXT
    frame:      POSITION
    orient_flag: BOOLEAN
BEGIN
    cur_pos = curpos(0,0) -- Get the current position of the robot
    cur_pos.x = 1000.00  -- Create a new position which is
                          -- at(1000,0,300,w,p,r)
    cur_pos.y = 0.0
```

```

cur_pos.z = 300.00
SET_EPOS_REG(1, cur_pos, status)
move_to_pr1 -- Call TP program to move to PR[1]
              -- The robot is now at a known position:
              --(1000,0,300,w,p,r) where (w,p,r) have not
              --changed from the original position.
pos_frame = curpos(0,0) -- Create a frame used to mirror about.
pos_frame.w = 0 -- By setting (w,p,r) to 0, the x-z plane of
pos_frame.p = 0 -- pos_frame will be parallel to the world's x-z
pos_frame.r = 0 -- plane. pos_frame now set to (1000,0,300,0,0,0)
frame = pos_frame -- Convert the mirror frame to a POSITION type.
cur_pos.y = 200 -- Move 200mm in the y direction.
SET_EPOS_REG(1, cur_pos, status)
move_to_pr1 -- Current position is (1000,200,300,w,p,r)
org_pos = cur_pos -- Convert org_pos to a POSITION type.
orient_flag = FALSE -- Send Mirror current position: (1000, 200,
                     -- 300, w, p, r), and mirror frame: (1000,0,
                     -- 300,0,0,0). Mirrors about the y axis
                     -- without mirroring the
                     -- orientation (w,p,r).
mir_pos = mirror(org_pos, frame, orient_flag)
          -- mir_pos is the mirrored position: (1000, -200,
          -- 300, w, p, r).
          -- The orientation is the same as org_pos.
orient_flag = TRUE -- The mirrored position includes mirroring
                     -- of the tool orientation.
mir_posa = mirror(org_pos, frame, orient_flag)
          -- mir_posa is the mirrored position where Normal
          -- Orient, & Approach vectors have been mirrored.
end mir_exam

```

**Figure A.13.1 MIRROR Built-In Function**

## A.13.2 MODIFY\_QUEUE Built-In Procedure

**Purpose:** Replaces the value of an entry of a queue.

**Syntax:** MODIFY\_QUEUE(value, sequence\_no, queue\_t, queue\_data, status)

Input/Output Parameters:

[in] value : INTEGER

[in] sequence\_no : INTEGER

[in,out] queue\_t : QUEUE\_TYPE

[in,out] queue\_data : ARRAY OF INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBQMGR

**Details:**

- value specifies the value to be inserted into the queue.
- sequence\_no specifies the sequence number of the entry whose value is to be modified
- queue\_t specifies the queue variable for the queue

- queue\_data specifies the array used to hold the data in the queue. The length of this array determines the maximum number of entries in the queue.
- status is returned with **61003**, Bad sequence no, if the specified sequence number is not in the queue.

**See Also:** [Section A.3.48, COPY\\_QUEUE Built-In Procedure](#) , [Section A.7.16, GET\\_QUEUE Built-In Procedure](#) , [Section A.4.14, DELETE\\_QUEUE Built-In Procedure](#) , [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#)

**Example:** In the following example, the routine update\_queue replaces the value of the specified entry (sequence\_no); of a queue (queue and queue\_data) with a new value (value).

```
PROGRAM mod_queue_x
%ENVIRONMENT PBQMGR
ROUTINE update_queue(value: INTEGER;
                      sequence_no: INTEGER;
                      queue_t: QUEUE_TYPE;
                      queue_data: ARRAY OF INTEGER)
VAR
    status: INTEGER
BEGIN
MODIFY_QUEUE(value, sequence_no, queue_t, queue_data, status)
return
END update_queue
BEGIN
END mod_queue_x
```

**Figure A.13.2 MODIFY\_QUEUE Built-In Procedure**

### A.13.3 MOTION\_CTL Built-In Function

**Purpose:** Determines whether the KAREL program has motion control for the specified group of axes

**Syntax:** MOTION\_CTL<(group\_mask)>

Function Return Type: BOOLEAN

Input/Output Parameters:

[in] group\_mask : INTEGER

%ENVIRONMENT Group : MOTN

**Details:**

- If group\_mask is omitted, the default group mask for the program is assumed.
- The default group\_mask is determined by the %LOCKGROUP and %NOLOCKGROUP directives.
- The group\_mask is specified by setting the bit(s) for the desired group(s).

**Table A.13.3 Group\_mask Setting**

GROUP	DECIMAL	BIT
Group 1	1	1
Group 2	2	2
Group 3	4	3

To specify multiple groups select the decimal values, shown in [Table A.13.3](#), which correspond to the desired groups. Then connect them together using the OR operator. For example to specify groups 1, 3, enter "1 OR 4".

- Returns TRUE if the KAREL program has motion control for the specified group of axes.

## A.13.4 MOUNT\_DEV Built-In Procedure

---

**Purpose:** Mounts the specified device

**Syntax:** MOUNT\_DEV (device, status)

Input/Output Parameters:

[in] device : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- device specifies the device to be mounted.
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** [Section A.4.26, DISMOUNT\\_DEV Built-In Procedure](#) , [Section A.6.6, FORMAT\\_DEV Built-In Procedure](#)

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#), for a detailed program example.

## A.13.5 MOVE\_FILE Built-In Procedure

---

**Purpose:** Moves the specified file from one memory file device to another

**Syntax:** MOVE\_FILE (file\_spec, status)

Input/Output Parameters:

[in] file\_spec : STRING

[out] status : INTEGER

%ENVIRONMENT Group :FDEV

**Details:**

- file\_spec specifies the device, name, and type of the file to be moved. The file should exist on the FROM or RAM disks.
- If file\_spec is a file on the FROM disk, the file is moved to the RAM disk, and vice versa.
- The wildcard character (\*) can be used to replace the entire file name, the first part of the file name, the last part of the file name, or both the first and last parts of the file name. The file type can also use the wildcard in the same manner. If file\_spec specifies multiple files, then they are all moved to the other disk.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** In the following example, all .KL files are moved from the RAM disk to the FDEV disk.

```
PROGRAM move_files
%NOLOCKGROUP
%ENVIRONMENT FDEV
VAR
    status: INTEGER
BEGIN
    MOVE_FILE('RD:\*.KL', status)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, 0)
    ENDIF
END move_files
```

**Figure A.13.5 MOVE\_FILE Built-In Procedure**

## A.13.6 MSG\_CONNECT Built-In Procedure

**Purpose:** Connect a client or server port to another computer for use in Socket Messaging.

**Syntax:** MSG\_CONNECT (tag, status)

Input/Output Parameters:

[in] tag : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FLBT

**Details:**

- tag is the name of a client port (C1:-C8) or server port (S1:-S8).
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** Socket Messaging in the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)*.

**Example:** The following example connects to S8: and reads messages. The messages are displayed on the teach pendant screen.

```
PROGRAM tcpserv8
VAR
    file_var : FILE
    tmp_int : INTEGER
    tmp_int1 : INTEGER
    tmp_str : string [128]
    tmp_str1 : string [128]
    status : integer
    entry : integer
BEGIN
    SET_FILE_ATR(file_var, ATR_IA)
    -- Set up S8 server tag
    DISMOUNT_DEV('S8:',status)
    MOUNT_DEV('S8:',status)
    write (' Mount Status = ',status,cr)
    status = 0
    IF status = 0 THEN
        -- Connect the tag
```

```

write ('Connecting ..',cr)
MSG_CONNECT ('S8:',status)
write ('Connect Status = ',status,cr)
IF status < > 0 THEN
    MSG_DISCO('S8:',status)
    write (' Connecting..',cr)
    MSG_CONNECT('S8:',status)
    write (' Connect Status = ',status,cr)
ENDIF
IF status = 0 THEN
    -- OPEN S8:
    write ('Opening',cr)
    OPEN FILE file_var ('rw','S8:')
    status = io_status(file_var)
    FOR tmp_int 1 TO 1000 DO
        write ('Reading',cr)
        BYTES_AHEAD(file_var, entry, status)
        -- Read 10 bytes
        READ file_var (tmp_str::10)
        status = i/o_status(file_var)
        --Write 10 bytes
        write (tmp_str::10,cr)
        status = io_status(file_var)
    ENDFOR
    CLOSE FILE file_var
    write ('Disconnecting..',cr)
    MSG_DISCO('S8:',status)
    write ('Done.',cr)
ENDIF
ENDIF
END tcperv8

```

**Figure A.13.6 MSG\_CONNECT Built-In Procedure**

## A.13.7 MSG\_DISCO Built-In Procedure

**Purpose:** Disconnect a client or server port from another computer.

**Syntax:** MSG\_DISCO (tag, status)

Input/Output Parameters:

[in] tag : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FLBT

**Details:**

- tag is the name of a client port (C1:-C8) or server port (S1:-S8).
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** Socket Messaging in the *Internet Options Setup and Operations Manual* (*MAROUIN9010171E*) or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)*.

**Example:** Refer to [Section A.13.6, MSG\\_CONNECT Built-In Procedure](#) for more examples.

## A.13.8 MSG\_PING Built-In Procedure

**Purpose:** Network utility that sends a request to a specific host name and expects a response.

**Syntax:** MSG\_PING (host name, status)

Input/Output Parameters:

[in] host name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FLBT

**Details:**

- Host name is the name of the host to perform the check on. An entry for the host has to be present in the host entry tables (or the DNS option loaded and configured on the robot).
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** Ping in the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)*.

**Example:** The following example performs a PING check on the hostname fido. It writes the results on the teach pendant.

```
PROGRAM pingtest
VAR
  Tmp_int    : INTEGER
  Status      : integer
BEGIN
  WRITE('pinging..',cr)
  MSG_PING('fido',status)
  WRITE('ping Status = ',status,cr)
END pingtest
```

Figure A.13.8 MSG\_PING Built-In Procedure

## A.14 - N - KAREL LANGUAGE DESCRIPTION

### A.14.1 NOABORT Action

**Purpose:** Prevents program execution from aborting when an external error occurs

**Syntax:** NOABORT

**Details:**

- The NOABORT action usually corresponds to an ERROR[n].
- If the program is aborted by itself (for example, executing an ABORT statement, run time error), the NOABORT action will be ignored and program execution will be aborted.

**Example:** The following example uses a global condition handler to test for error number **11038**, Pulse Mismatch. If this error occurs, the NOABORT action will prevent program execution from being aborted.

```
PROGRAM noabort_ex
%NOLOCKGROUP
BEGIN
    --Pulse Mismatch condition handler
    CONDITION[801]:
        WHEN ERROR[11038] DO
            NOABORT
    ENDCONDITON
    ENABLE CONDITION[801]
END noabort_ex
```

**Figure A.14.1 NOABORT Action**

## A.14.2 %NOABORT Translator Directive

---

**Purpose:** Specifies a mask for aborting

**Syntax:** %NOABORT = ERROR + COMMAND

**Details:**

- ERROR and COMMAND are defined as follows:
  - ERROR – ignore abort error severity
  - COMMAND – ignore abort command
- Any combination of ERROR and COMMAND can be specified.
- If the program is aborted by itself (for example, executing an ABORT statement, run-time error), the %NOABORT directive will be ignored and program execution will be aborted.
- This directive is only effective for programs with %NOLOCKGROUP. If the program has motion control, the %NOABORT directive will be ignored and program execution will be aborted.

## A.14.3 %NOBUSYLAMP Translator Directive

---

**Purpose:** Specifies that the busy lamp will be OFF during execution.

**Syntax:** %NOBUSYLAMP

**Details:**

- The busy lamp can be set during task execution by the SET\_TSK\_ATTR built-in.

## A.14.4 NODE\_SIZE Built-In Function

---

**Purpose:** Returns the size (in bytes) of a PATH node

**Syntax:** NODE\_SIZE(path\_var)

Function Return Type: INTEGER

Input/Output Parameters:

[in] path\_var : PATH

%ENVIRONMENT Group : PATHOP

**Details:**

- The returned value is the size of an individual PATH node, including the positional data type size and any associated data.
- The returned value can be used to calculate file positions for random access to nodes in files.

**Example:** The following example program reads a path, while overlapping reads with preceding moves. The routine read\_header reads the path header and prepares for reading of nodes. The routine read\_node reads a path node.

```

PROGRAM read_and_mov
VAR my_path: PATH
    xyz_pos: XYZWPR
    path_base: INTEGER
    node_size: INTEGER
    max_node_no: INTEGER
    i: INTEGER
    file_var: FILE
--
ROUTINE read_header
BEGIN
    READ file_var(my_path[0])
    IF IO_STATUS(file_var) <> 0 THEN
        WRITE('HEADER READ ERROR:',IO_STATUS(file_var),cr)
        ABORT
    ENDIF
    max_node_no = PATH_LEN(my_path)
    node_size = NODE_SIZE(my_path)
    path_base = GET_FILE_POS(file_var)
END read_header
--
ROUTINE read_node(node_no: INTEGER)
VAR status: INTEGER
BEGIN
    SET_FILE_POS(file_var, path_base+(node_no-1)*node_size, status)
    READ file_var(my_path[node_no])
END read_node
--
BEGIN
    SET_FILE_ATR(file_var, atr_uf)
    OPEN FILE F1('RO','PATHFILE.DT')
    read_header
    FOR i = 1 TO max_node_no DO
        read_node(i)
        xyz_pos = my_path[i]
        SET_POS_REG(1 xyz_pos, status)
        move_to_pr1 — Call TP program to move to node
    ENDFOR

```

```
CLOSE FILE file_var
END read_and_mov
```

**Figure A.14.4 NODE\_SIZE Built-In Function**

## A.14.5 %NOLOCKGROUP Translator Directive

---

**Purpose:** Specifies that motion groups do not need to be locked when calling this program, or a routine defined in this program.

**Syntax:** %NOLOCKGROUP

**Details:**

- When the program or routine is called, the task will not attempt to get motion control.
- If %NOLOCKGROUP is not specified, all groups will be locked when the program or routine is called, and the task will attempt to get motion control. The task will pause if it cannot get motion control.
- The task will keep motion control while it is executing the program or routine. When it exits the program or routine, the task automatically unlocks all the motion groups.
- If the task contains executing or stopped motion, then task execution is held until the motion is completed. Stopped motion must be resumed and completed or canceled.
- If a program that has motion control calls a program with the %NOLOCKGROUP Directive or a routine defined in such a program, the program will keep motion control even though it is not needed.
- The UNLOCK\_GROUP built-in routine can be used to release control.
- If a motion statement is encountered in a program that has the %NOLOCKGROUP directive, the task will attempt to get motion control for all the required groups if it does not already have it. The task will pause if it cannot get motion control.

**See Also:** [Section A.12.5, %LOCKGROUP Translator Directive](#) , [Section A.12.4, LOCK\\_GROUP Built-In Procedure](#) , [Section A.21.5, UNLOCK\\_GROUP Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

## A.14.6 NOMESSAGE Action

---

**Purpose:** Suppresses the display and logging of error messages

**Syntax:** NOMESSAGE

**Details:**

- Display and logging of the error messages are suppressed only for the error number specified in the corresponding condition.
- Use a wildcard (\*) to suppress all messages.
- Abort error messages still will be displayed and logged even if NOMESSAGE is used.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.14.7 NOPAUSE Action

**Purpose:** Resumes program execution if the program was paused, or prevents program execution from pausing

**Syntax:** NOPAUSE

**Details:**

- The NOPAUSE action usually corresponds to an ERROR[n] or PAUSE condition.
- The program will be resumed, even if it was paused before the error.
- If the program is paused by itself, the NOPAUSE action will be ignored and program execution will be paused.

**Example** The following example uses a global condition handler to test for error number **12311**. If this error occurs, the NOPAUSE action will prevent program execution from being paused and the NOMESSAGE action will suppress the error message normally displayed for error number **12311**. This will allow the routine `uninit_error` to be executed without interruption.

```
ROUTINE uninit_error
BEGIN
    WRITE ('Uninitialized operand',CR)
    WRITE ('Use KCL> SET VAR to initialize operand',CR)
    WRITE ('Press Resume at Test/Run screen to ',cr)
    WRITE ('continue program',cr)
    PAUSE --pauses program (undoes NOPAUSE action)
END uninit_error
CONDITION[1]:
WHEN ERROR[12311] DO
    NOPAUSE, NOMESSAGE, uninit_error
ENDCONDITION
```

**Figure A.14.7 NOPAUSE Action**

## A.14.8 %NOPAUSE Translator Directive

**Purpose:** Specifies a mask for pausing

**Syntax:** %NOPAUSE = ERROR + COMMAND + TPENABLE

**Details:**

- The bits for the mask are as follows:
  - ERROR – ignore pause error severity

- COMMAND – ignore pause command
- TPENABLE – ignore paused request when TP enabled
- Any combination of ERROR, COMMAND, and TPENABLE can be specified.
- If the program is paused by itself, the %NOPAUSE directive will be ignored and program execution will be paused.
- This directive is only effective for programs with %NOLOCKGROUP. If the program has motion control, the %NOPAUSE directive will be ignored and program execution will be paused.

## A.14.9 %NOPAUSESHFT Translator Directive

---

**Purpose:** Specifies that the task is not paused if shift key is released.

**Syntax:** %NOPAUSESHFT

**Details:**

- This attribute can be set during task execution by the SET\_TSK\_ATTR built-in routine.

## A.15 - O - KAREL LANGUAGE DESCRIPTION

---

### A.15.1 OPEN FILE Statement

---

**Purpose:** Associates a data file or communication port with a file variable

**Syntax:** OPEN FILE file\_var (usage\_string, file\_string)

where:

file\_var : FILE

usage\_string : a STRING

file\_string : a STRING

**Details:**

- file\_var must be a static variable not already in use by another OPEN FILE statement.
- The usage\_string is composed of the following:
  - 'RO': Read only
  - 'RW': Read write
  - 'AP': Append
  - 'UD': Update
- The file\_string identifies a data file name and type, a window or keyboard, or a communication port.
- The SET\_FILE\_ATTR built-in routine can be used to set a file's attributes.
- When a program is aborted or exits normally, any opened files are closed. Files are not closed when a program is paused.
- Use the IO\_STATUS built-in function to verify if the open file operation was successful.

**See Also:** [Section A.9.19, IO\\_STATUS Built-In Function](#) , [Section A.19.13, SET\\_FILE\\_ATR Built-In Procedure](#) , [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#) , [Chapter 10, FILE SYSTEM](#) , [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

## A.15.2 OPEN HAND Statement

**Purpose:** Opens a hand on the robot

**Syntax:** OPEN HAND hand\_num

where:

hand\_num : an INTEGER expression

**Details:**

- The actual effect of the statement depends on how the HAND signals are set up. Refer to [Chapter 16, INPUT/OUTPUT SYSTEM](#)
- hand\_num must be a value in the range 1-2. Otherwise, the program is aborted with an error.
- The statement has no effect if the value of hand\_num is in range but the hand is not connected.
- If the value of hand\_num is in range but the HAND signal represented by that value has not been assigned, the program is aborted with an error.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) , for more syntax information

**Example:** The following example moves the TCP to the position register PR[2] and opens the hand of the robot specified by the INTEGER variable hand\_num.

```
move_to_pr2 -- Call TP program to move to PR[2]
OPEN HAND hand_num
```

**Figure A.15.2 OPEN HAND Statement**

## A.15.3 OPEN\_TPE Built-In Procedure

**Purpose:** Opens the specified teach pendant program

**Syntax:** OPEN\_TPE(prog\_name, open\_mode, reject\_mode, open\_id, status)

Input/Output Parameters:

[in] prog\_name : STRING

[in] open\_mode : INTEGER

[in] reject\_mode : INTEGER

[out] open\_id : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

#### Details:

- prog\_name specifies the name of the teach pendant program to be opened. prog\_name must be in all capital letters.
- prog\_name must be closed, using CLOSE\_TPE, before prog\_name can be executed.
- open\_mode determines the access code to the program. The access codes are defined as follows:
  - 0 : none
  - TPE\_RDACC : Read Access
  - TPE\_RWACC : Read/Write Access
- reject\_mode determines the reject code to the program. The program that has been with a reject code cannot be opened by another program. The reject codes are defined as follows:
  - TPE\_NOREJ : none
  - TPE\_RDREJ : Read Reject
  - TPE\_WRTREJ : Write Reject
  - TPE\_RWREJ : Read/Write Reject
  - TPE\_ALLREJ : All Reject
- open\_id indicates the ID number of the opened program.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.
- All open teach pendant programs are closed automatically when the KAREL program is aborted or exits normally.

**See Also:** [Section A.3.52, CREATE\\_TPE Built-In Procedure](#), [Section A.3.49, COPY\\_TPE Built-In Procedure](#), [Section A.1.27, AVL\\_POS\\_NUM Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

---

## A.15.4 ORD Built-In Function

---

**Purpose:** Returns the numeric ASCII code corresponding to the character in the STRING argument that is referenced by the index argument

**Syntax:** ORD(str, str\_index)

Function Return Type : INTEGER

Input/Output Parameters:

[in] str : STRING

[in] str\_index : INTEGER

%ENVIRONMENT Group : SYSTEM

#### Details:

- The returned value represents the ASCII numeric code of the specified character.
- str\_index specifies the indexed position of a character in the argument str. A value of 1 indicates the first character.
- If str\_index is less than one or greater than the current length of str, the program is paused with an error.

**See Also:** [Appendix D, CHARACTER CODES](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.15.5 ORIENT Built-In Function

---

**Purpose:** Returns a unit VECTOR representing the y-axis (orient vector) of the specified POSITION argument

**Syntax:** ORIENT(posn)

Function Return Type: VECTOR

Input/Output Parameters:

[in] posn : POSITION

%ENVIRONMENT Group : VECTR

**Details:**

- Instead of using this built-in, you can directly access the Orient Vector of a POSITION.
- The returned value is the orient vector of posn.
- The orient vector is the positive y-direction in the tool coordinate frame.

## A.16 - P - KAREL LANGUAGE DESCRIPTION

---

### A.16.1 PATH Data Type

---

**Purpose:** Defines a variable or routine parameter as PATH data type

**Syntax:** PATH

**Details:**

- A PATH is a varying length list of elements called path nodes, numbered from 1 to the number of nodes in the PATH.
- No valid operators are defined for use with PATH variables.
- A PATH variable is indexed (or subscripted) as if it were an ARRAY variable. For example, `tool_track[1]` refers to the first node of a PATH called `tool_track`.
- An uninitialized PATH has a length of zero.
- PATH variables cannot be declared local to routines and cannot be returned from functions.
- Only PATH expressions can be assigned to PATH variables or passed as arguments to PATH parameters.
- A PATH variable can specify a data structure constituting the data for each path node.
- A PATH variable can specify a data structure constituting the path header. This can be used to specify the UFRAME and/or UTOOL to be used with recording the path. It can also specify an axis group whose current position defines a table-top coordinate frame with respect to which the robot data is recorded.

- A PATH can be declared with either, neither, or both of the following clauses following the word PATH:
  - NODEDATA = node\_struct\_name, specifying the data structure constituting a path node.
  - PATHHEADER = header\_struct\_name, specifying the structure constituting the path header.

If both fields are present, they can appear in either order and are separated by a comma and optionally a new line.

- If NODEDATA is not specified, it defaults to the STD\_PTH\_NODE structure described in [Section A.19.47, STD\\_PTH\\_NODE Data Type](#).
- If PATHHEADER is not specified, there is no (user-accessible) path header.
- An element of the PATHHEADER structure can be referenced with the syntax path\_var\_name.header\_field\_name.
- An element of a NODEDATA structure can be referenced with the syntax path\_var\_name[node\_no].node\_field\_name.
- The path header structure can be copied from one path to another with the path\_var1 = path\_var2 statement.
- The path node structure can be copied from one node to another with the path\_var[2] = path\_var[1] statement.
- A path can be passed as an argument to a routine as long as the PATHHEADER and NODEDATA types match. A path that is passed as an argument to a built-in routine can be of any type.
- A path node can be passed as an argument to a routine as long as the routine parameter is the same type as the NODEDATA structure.
- A path can be declared with a NODEDATA structure having no position type elements. This can be a useful way of maintaining a list structure.

**See Also:** [Section A.1.15, APPEND\\_NODE Built-In Procedure](#), [Section A.4.13, DELETE\\_NODE Built-In Procedure](#), [Section A.9.14, INSERT\\_NODE Built-In Procedure](#), [Section A.16.2, PATH\\_LEN Built-In Function](#), [Section A.14.4, NODE\\_SIZE Built-In Function](#)

**Example:** The following example shows different declarations of PATH variables.

```

TYPE
    node_struct = STRUCTURE
        node_posn: XYZWPR IN GROUP[1]
        aux_posn: JOINTPOS IN GROUP[2]
        weld_time: INTEGER
        weld_current: INTEGER
    ENDSTRUCTURE
    hdr_struct = STRUCTURE
        uframe1: POSITION
        utool: POSITION
        speed: REAL
    ENDSTRUCTURE
VAR
    path_1a: PATH PATHHEADER = hdr_struct, NODEDATA = node_struct
    path_1b: PATH NODEDATA = node_struct, PATHHEADER = hdr_struct
    path_2: PATH NODEDATA = node_struct -- no header
    path_3: PATH -- NODEDATA is STD_PTH_NODE
    path_4: PATH PATHHEADER = hdr_struct -- NODEDATA is
STD_PTH_NODE

```

**Figure A.16.1 (a) PATH Data Type**

The following example shows how elements of the NODEDATA and PATHHEADER structures can be referenced.

```
-- Using declarations for path_1a:  
-- Using NODEDATA fields:  
path_1a[1].node_posn = CURPOS(0, 0)  
cnt_dn_time = path_1a[node_no].weld_time  
-- Using PATHHEADER fields:  
path_1a.utool = tool_1
```

**Figure A.16.1 (b) PATH Data Type**

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM](#)

## A.16.2 PATH\_LEN Built-In Function

**Purpose:** Returns the number of nodes in the PATH argument

**Syntax:** PATH\_LEN(path\_nam)

Function Return Type : INTEGER

Input/Output Parameters:

[in] path\_nam : PATH

%ENVIRONMENT Group : PBCORE

**Details:**

- The returned value corresponds to the number of nodes in the PATH variable argument.
- Calling PATH\_LEN with an uninitialized PATH returns a value of zero.

**See Also:** [Section A.3.47, COPY\\_PATH Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.16.3 PAUSE Action

**Purpose:** Suspends execution of a running task

**Syntax:** PAUSE <PROGRAM[n]>

**Details:**

- The PAUSE action pauses task execution in the following manner:

- Any motion already initiated continues until completed.
- Files are left open.
- All connected timers continue being incremented.
- All PULSE statements in execution continue execution.
- Sensing of conditions specified in condition handlers continues.
- Any actions, except routine call actions, are completed. Routine call actions are performed when the program is resumed.
- The PAUSE action can be followed by the clause PROGRAM[n], where n is the task number to be paused.
- Use GET\_TSK\_INFO to find a task number.

**See Also:** [Section A.21.6, UNPAUSE Action](#)

## A.16.4 PAUSE Condition

---

**Purpose:** Monitors the pausing of program execution

**Syntax:** PAUSE < PROGRAM [n] >

**Details:**

- The PAUSE condition is satisfied when a program is paused, for example, by an error, a PAUSE Statement, or the PAUSE action.
  - If one of the actions corresponding to a PAUSE condition is a routine call, it is necessary to specify a NOPAUSE action to allow execution of the routine.
- Also, the routine being called needs to include a PAUSE statement so the system can handle completely the cause of the original pause.
- The PAUSE condition can be followed by the clause PROGRAM[n], where n is the task number to be paused.
  - Use GET\_TSK\_INFO to find a task number.

**Example:** The following example scans for the PAUSE condition in a global condition handler. If this condition is satisfied, DOUT[1] will be turned on. The CONTINUE action continues program execution; ENABLE re-enables the condition handler.

```
CONDITION[1] :
  WHEN PAUSE DO
    DOUT[1] = TRUE
    CONTINUE
    ENABLE CONDITION[1]
ENDCONDITION
```

**Figure A.16.4 PAUSE Condition**

## A.16.5 PAUSE Statement

---

**Purpose:** Suspends execution of a KAREL program

**Syntax:** PAUSE < PROGRAM [n] >

**Details:**

- The PAUSE statement pauses program execution in the following manner:
  - Any motion already initiated continues until completed.
  - Files are left open.
  - All connected timers continue being incremented.
  - All PULSE statements in execution continue execution.
  - Sensing of conditions specified in condition handlers continues.
  - Any actions, except routine call actions, are completed. Routine call actions are performed when the program is resumed.
- The PAUSE statement can be followed by the clause PROGRAM[n], where n is the task number to be paused.
- Use GET\_TSK\_INFO to find a task number.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** If DIN[1] is TRUE, the following example pauses the KAREL program using the PAUSE statement. The message, Program is paused. Press RESUME function key to continue will be displayed on the CRT/KB screen.

```
PROGRAM p_pause
BEGIN
    IF DIN[1] THEN
        WRITE ('Program is Paused. ')
        WRITE ('Press RESUME function key to continue', CR)
        PAUSE
    ENDIF
END p_pause
```

**Figure A.16.5 PAUSE Statement**

## A.16.6 PAUSE\_TASK Built-In Procedure

**Purpose:** Pauses the specified executing task

**Syntax:** PAUSE\_TASK(task\_name, force\_sw, stop\_mtn\_sw, status)

Input/Output Parameters:

[in] task\_name : STRING  
 [in] force\_sw : BOOLEAN  
 [in] stop\_mtn\_sw : BOOLEAN  
 [out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- task\_name is the name of the task to be paused. If task name is '\*ALL\*', all executing tasks are paused except the tasks that have the ignore pause request attribute set.
- force\_sw specifies whether a task should be paused even if the task has the ignore pause request attribute set. This parameter is ignored if task\_name is '\*ALL\*'.
- stop\_mtn\_sw specifies whether all motion groups belonging to the specified task are stopped.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.18.43, RUN\\_TASK Built-In Procedure](#), [Section A.3.45, CONT\\_TASK Built-In Procedure](#), [Section A.1.4, ABORT\\_TASK Built-In Procedure](#), [Chapter 17, MULTI-TASKING](#)

**Example:** The following example pauses the user-specified task and stops any motion. Refer to [Chapter 17, MULTI-TASKING](#) for more examples.

```
PROGRAM pause_ex
%ENVIRONMENT MULTI
VAR
    task_str: STRING[12]
    status : INTEGER
BEGIN
    WRITE('Enter task name to pause:')
    READ(task_str)
    PAUSE_TASK(task_str, TRUE, TRUE, status)
END pause_ex
```

**Figure A.16.6 PAUSE\_TASK Built-In Procedure**

## A.16.7 PEND\_SEMA Built-In Procedure

**Purpose:** Suspends execution of the task until either the value of the semaphore is greater than zero or max\_time expires

**Syntax:** PEND\_SEMA(semaphore\_no, max\_time, time\_out)

Input/Output Parameters:

[in] semaphore\_no : INTEGER

[in] max\_time : INTEGER

[out] time\_out : BOOLEAN

%ENVIRONMENT Group : MULTI

### Details:

- PEND\_SEMA decrements the value of the semaphore.
- semaphore\_no specifies the semaphore number to use.
- semaphore\_no must be in the range of 1 to the number of semaphores defined on the controller.
- max\_time specifies the expiration time, in milliseconds. A max\_time value of -1 indicates to wait forever, if necessary.
- On continuation, time\_out is set TRUE if max\_time expired without the semaphore becoming nonzero, otherwise it is set FALSE.

**See Also:** [Section A.16.18, POST\\_SEMA Built-In Procedure](#), [Section A.3.11, CLEAR\\_SEMA Built-In Procedure](#), [Section A.19.5, SEMA\\_COUNT Built-In Function](#), [Chapter 17, MULTI-TASKING](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.16.8 PIPE\_CONFIG Built-In Procedure

---

**Purpose:** Configure a pipe for special use.

**Syntax:** PIPE\_CONFIG(pipe\_name, cmos\_flag, n\_sectors, record\_size, form\_dict, form\_ele, status)

Input/Output Parameters :

[in] pipe\_name : STRING

[in] cmos\_flag : BOOLEAN

[in] n\_sectors : INTEGER

[in] record\_size : INTEGER

[in] form\_dict : STRING

[in] form\_ele : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : FLBT

**Details:**

- pipe\_name is the name of the pipe file. If the file does not exist it will be created with this operation.
- cmos\_flag if set to TRUE will put the pipe data in CMOS. By default pipe data is in DRAM.
- n\_sectors number of 1024 byte sectors to allocate to the pipe. The default is 8.
- record\_size the size of a binary record in a pipe. If set to 0 the pipe is treated as ASCII. If a pipe is binary and will be printed as a formatted data then this must be set to the record length.
- form\_dict is the name of the dictionary containing formatting information.
- form\_ele is the element number in form\_dict containing the formatting information.
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

**See Also:** For further information see PIP: Device in [Section 10.2.4, File Pipes](#).

## A.16.9 POP\_KEY\_RD Built-In Procedure

---

**Purpose:** Resumes key input from a keyboard device

**Syntax:** POP\_KEY\_RD(key\_dev\_name, pop\_index, status)

Input/Output Parameters:

[in] key\_dev\_name : STRING

[in] pop\_index : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Resumes all suspended read requests on the specified keyboard device.
- If there were no read requests active when suspended, this operation will not resume any inputs. This is not an error.
- key\_dev\_name must be one of the keyboard devices already defined:

- 'TPKB': Teach Pendant Keyboard Device
- 'CRKB': CRT Keyboard Device
- pop\_id is returned from PUSH\_KEY\_RD and should be used to re-activate the read requests.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.16.30, PUSH\\_KEY\\_RD Built-In Procedure](#), [Section A.18.4, READ\\_KB Built-In Procedure](#)

**Example:** Refer to the example for the READ\_KB built-in routine.

## A.16.10 Port\_Id Action

---

**Purpose:** Sets the value of a port array element to the result of an expression

**Syntax:** Port\_Id[n] = expression

where:

Port\_Id : an output port array

n : an INTEGER

expression : a variable, constant, or EVAL clause

**Details:**

- The value of expression is assigned to the port array element referenced by n.
- The port array must be an output port array that can be written to by a KAREL program. Refer to [Chapter 2, LANGUAGE ELEMENTS](#).
- expression can be a user-defined, static variable, a system variable that can be read by a KAREL program, any constant, or an EVAL clause.
- If expression is a variable, the value used is the current value of the variable at the time the action is taken, not when the condition handler is defined.
- If expression is an EVAL clause, it is evaluated when the condition handler is defined and that value is assigned when the action is taken.
- The expression must be of the same type as Port\_Id.
- You cannot assign a port array element to a port array element directly.
- If the expression is a variable that is uninitialized when the condition handler is enabled, the program will be aborted with an error.

**See Also:** [Chapter 6, CONDITION HANDLERS](#), [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#), [Section A.18.6, Relational Condition](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.16.11 Port\_Id Condition

---

**Purpose:** Monitors a digital port signal

**Syntax:** <NOT> Port\_Id[n] < + | - >

where:

Port\_Id : a port array

n : an INTEGER

**Details:**

- n specifies the port array signal to be monitored.
- Port\_Id must be one of the predefined BOOLEAN port array identifiers with read access. Refer to [Chapter 2, LANGUAGE ELEMENTS](#).
- For event conditions, only the + or - alternatives are used.
- For state conditions, only the NOT alternative is used.

**See Also:** [Chapter 6, CONDITION HANDLERS](#), [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#), [Section A.18.6, Relational Condition](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING](#) (DOUT\_EX.KL) for a detailed program example.

## A.16.12 POS Built-In Function

**Purpose:** Returns an XYZWPR composed of the specified location arguments (x,y,z), orientation arguments (w,p,r), and configuration argument (c)

**Syntax:** POS(x, y, z, w, p, r, c)

Function Return Type: XYZWPR

Input/Output Parameters:

[in] x, y, z, w, p, and r : REAL

[in] c : CONFIG

%ENVIRONMENT Group : SYSTEM

**Details:**

- c must be a valid configuration for the robot attached to the controller. CNV\_STR\_CONF can be used to convert a string to a CONFIG variable.
- x, y, and z are the Cartesian values of the location (in millimeters). Each argument must be in the range  $\pm 10000000$  mm ( $\pm 10$  km). Otherwise, the program is paused with an error.
- w, p, and r are the yaw, pitch, and roll values of the orientation (in degrees). Each argument must be in the range  $\pm 180$  degrees. Otherwise, the program is paused with an error.

**See Also:** [Chapter 8, POSITION DATA](#)

**Example:** The following example uses the POS built-In to designate numerically the POSITION next\_pos.

```
CNV_STR_CONF('n', config_var, status)
next_pos = POS(100,-400.25,0.5,10,-45,30,config_var)
```

**Figure A.16.12 POS Built-In Function**

## A.16.13 POS2JOINT Built-In Function

---

**Purpose:** This routine is used to convert Cartesian positions (in\_pos) to joint angles (out\_jnt) by calling the inverse kinematics routine.

**Syntax:** POS2JOINT (ref\_jnt, in\_pos, uframe, utool, config\_ref, wjnt\_cfg, ext\_ang, out\_jnt, status)

Input/Output Parameters:

[in] ref\_jnt : JOINTPOS  
 [in] in\_pos : POSITION  
 [in] uframe : POSITION  
 [in] utool : POSITION  
 [in] config\_ref : INTEGER  
 [in] wjnt\_cfg : CONFIG  
 [in] ext\_ang : ARRAY OF REAL  
 [out] out\_jnt : JOINTPOS  
 [out] status : INTEGER

%ENVIRONMENT Group: MOTN

**Details:**

- The input ref\_jnt are the reference joint angles that represent the robot's position just before moving to the current position.
- The input in\_pos is the robot Cartesian position to be converted to joint angles.
- The input uframe is the user frame for the Cartesian position.
- The input utool is the corresponding tool frame.
- The input config\_ref is an integer representing the type of solution desired. The values listed below are valid. Also, the pre-defined constants in the parentheses can be used and the values can be added as required. One example includes: config\_ref=HALF\_SOLN+CONFIG\_TCP.
  - 0 : (FULL\_SOLN) = Default
  - 1 : (HALF\_SOLN) = Wrist joint (XYZ456). This does not calculate/use WPR.
  - 2 : (CONFIG\_TCP) = The Wrist Joint Config (up/down) is based on the fixed wrist.
  - 4 : (APPROX\_SOLN) = Approximate solution. Reduce calculation time for some robots.
  - 8 : (NO\_TURNS) = Ignore wrist turn numbers. Use the closest path for joints 4, 5 and 6 (uses ref\_jnt).
  - 16 : (NO\_M\_TURNS) = Ignore major axis (J1 only) turn number. Use the closest path.
- The input wjnt\_cfg is the wrist joint configuration. This value must be input when config\_ref corresponds to HALF\_SOLN.
- The input ext\_ang contains the values of the joint angles for the extended axes if they exist.
- The output out\_jnt are the joint angles that correspond to the Cartesian position
- The output status explains the status of the attempted operation. If it is not equal to 0, then an error has occurred.

## A.16.14 POS\_REG\_TYPE Built-In Procedure

**Purpose:** Returns the position representation of the specified position register

**Syntax:** POS\_REG\_TYPE (register\_no, group\_no, posn\_type, num\_axes, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] group\_no : INTEGER

[out] posn\_type : INTEGER

[out] num\_axes : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register.
- If group\_no is omitted, the default group for the program is assumed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.
- posn\_type returns the position type. posn\_type is defined as follows:
  - 1 : POSITION
  - 2 : XYZWPR
  - 6 : XYZWPREXT
  - 9 : JOINTPOS
- num\_axes returns number of axes in the representation if the position type is a JOINTPOS. If the position type is an XYZWPREXT, only the number of extended axes is returned by num\_axes.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.12, GET\\_POS\\_REG Built-In Function](#) , [Section A.7.3, GET\\_JPOS\\_REG Built-In Function](#) , [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#) , [Section A.19.16, SET\\_JPOS\\_REG Built-In Procedure](#)

**Example:** The following example determines the position type in the register and uses the appropriate built-in to get data.

```
PROGRAM get_reg_data
%NOLOCKGROUP
%ENVIRONMENT REGOPE
VAR
    entry: INTEGER
    group_no: INTEGER
    jpos: JOINTPOS
    maxpregnum: integer
    num_axes: INTEGER
    posn_type: INTEGER
    register_no: INTEGER
    status: INTEGER
    xyz: XYZWPR
    xyzext: XYZWPREXTBEGIN
    group_no = 1
    GET_VAR(entry, '*POSREG*', '$MAXPREGNUM', maxpregnum, status)
    -- Loop for each register
```

```

FOR register_no = 1 to 10 DO
    -- Get the position register type
    POS_REG_TYPE(register_no, group_no, posn_type, num_axes, status)
    -- Get the position register
    WRITE('PR[', register_no, '] of type ', posn_type, CR)
    SELECT posn_type OF
        CASE (2):
            xyz = GET_POS_REG(register_no, status)
        CASE (6):
            xyzext = GET_POS_REG(register_no, status)
        CASE (9):
            jpos = GET_JPOS_REG(register_no, status)
        ELSE:
    ENDSELECT
ENDFOR
END get_reg_data

```

**Figure A.16.14 POS\_REG\_TYPE Built-In Procedure**

## A.16.15 POSITION Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as POSITION data type

**Syntax:** POSITION <IN GROUP[n]>

**Details:**

- A POSITION consists of a matrix defining the normal, orient, approach, and location vectors and a component specifying a configuration string, for a total of 60 bytes.
- The configuration string indicates the joint placements and multiple turns that describe the configuration of the robot when it is at a particular position.
- A POSITION is always referenced with respect to a specific coordinate frame.
- The POSITION data type can be used to represent a frame of reference in which case the configuration component is ignored.
- Coordinate frame transformations can be done using the relative position operator (:).
- A POSITION can be assigned to other positional types.
- Valid POSITION operators are the
  - Relative position (:) operator
  - Approximately equal (>=<) operator
- A POSITION can be followed by IN GROUP[n], where n indicates the motion group with which the data is to be used. The default is the group specified by the %DEFGROUP directive, or 1.
- Components of POSITION variables can be accessed or set as if they were defined as follows:

```

POSITION = STRUCTURE
    NORMAL: VECTOR          -- read-only
    ORIENT: VECTOR           -- read-only
    APPROACH: VECTOR         -- read-only
    LOCATION: VECTOR         -- read-write
    CONFIG_DATA: CONFIG      -- read-write
ENDSTRUCTURE

```

**Figure A.16.15 POSITION Data Type**

**See Also:** [Section A.16.12, POS Built-In Function](#) , [Section A.21.7, UNPOS Built-In Procedure](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.16.16 POST\_ERR Built-In Procedure

**Purpose:** Posts the error code and reason code to the error reporting system to display and keep history of the errors

**Syntax:** POST\_ERR(error\_code, parameter, cause\_code, severity)

Input/Output Parameters:

[in] error\_code : INTEGER

[in] parameter : STRING

[in] cause\_code : INTEGER

[in] severity : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- error\_code is the error to be posted.
- parameter will be included in error\_code's message if %s is specified in the dictionary text. If not necessary, then enter the null string.
- cause\_code is the reason for the error. 0 can be used if no cause is applicable.
- error\_code and cause\_code are in the following format: ffccc (decimal)

where

ff represents the facility code of the error.

ccc represents the error code within the specified facility.

- severity is defined as follows:

- 0 : WARNING, no change in task execution
- 1 : PAUSE, all tasks and stop all motion
- 2 : ABORT, all tasks and cancel

**See Also:** [Section A.5.4, ERR\\_DATA Built-In Procedure](#), the *Error Code Manual (MARRUEROR02171E)* or the *OPERATOR'S MANUAL (Alarm Code List) (B-83284EN)*

**Example:** Refer to [Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#), for a detailed program example.

## A.16.17 POST\_ERR\_L Built-In Procedure

**Purpose:** Posts the error code with local severity to the error reporting system to display and keep history of the errors

**Syntax:** POST\_ERR\_L(error\_code, parameter, cause\_code, severity)

Input/Output Parameters:

[in] error\_code : INTEGER

[in] parameter : STRING  
 [in] cause\_code : INTEGER  
 [in] severity : INTEGER  
 %ENVIRONMENT Group : PBCORE

**Details:**

- error\_code is the error to be posted.
- parameter will be included in error\_code's message if %s is specified in the dictionary text. The string will be the 7th parameter because 6 other parameters are passed internally. If parameter is not necessary, then enter the null string.
- cause\_code is the reason for the error. 0 can be used if no cause is applicable.
- error\_code and cause\_code are in the following format: ffccc (decimal)

where

ff represents the facility code of the error.

ccc represents the error code within the specified facility.

- severity is defined as follows:

- ERSEV\_NONE : No severity
- ERSEV\_WARN : WARNING, no change in task execution
- ERSEV\_PAUSE : PAUSE Global, pause all tasks and stop all motion after current motion segment
- ERSEV\_PAUSL : PAUSE Local, pause local task and stop all motion for local task after current motion segment
- ERSEV\_STOP : STOP Global, pause all tasks and stop all motion
- ERSEV\_STOPL : STOP Local, pause local task and stop all motion for local task
- ERSEV\_SERVO : SERVO Global, turn off all servo power and pause all tasks
- ERSEV\_SERVOL : SERVO Local, turn off servo power for local task motion groups and pause local tasks
- ERSEV\_ABORT : ABORT Global, abort all tasks and cancel all motion
- ERSEV\_ABORTL : ABORT Local, abort local task and cancel all motion for local task
- ERSEV\_SYSTEM : SYSTEM Global, system problem exist and prevent any further operation

**See Also:** [Section A.16.16, POST\\_ERR Built-In Procedure](#)

## A.16.18 POST\_SEMA Built-In Procedure

**Purpose:** Add one to the value of the indicated semaphore

**Syntax:** POST\_SEMA(semaphore\_no)

Input/Output Parameters:

[in] semaphore\_no : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- The semaphore indicated by semaphore\_no is incremented by one.
- semaphore\_no must be in the range of 1 to the number of semaphores defined on the controller.

**See Also:** [Section A.16.7, PEND\\_SEMA Built-In Procedure](#) , [Section A.3.11, CLEAR\\_SEMA Built-In Procedure](#) , [Section A.19.5, SEMA\\_COUNT Built-In Function](#) , [Chapter 17, MULTI-TASKING](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.16.19 PRINT\_FILE Built-In Procedure

**Purpose:** Prints the contents of an ASCII file to the default device

**Syntax:** PRINT\_FILE(file\_spec, nowait\_sw, status)

Input/Output Parameters:

[in] file\_spec : STRING  
[in] nowait\_sw : BOOLEAN  
[out] status : INTEGER  
%ENVIRONMENT Group : FDEV

**Details:**

- file\_spec specifies the device, name, and type of the file to print.
- If nowait\_sw is TRUE, execution of the program continues while the command is executing. If it is FALSE, the program stops, including condition handlers, until the operation has completed. If you have time critical condition handlers in your program, put them in another program that executes as a separate task.

### NOTE

nowait\_sw is not available in this release and should be set to FALSE.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

## A.16.20 %PRIORITY Translator Directive

**Purpose:** Specifies task priority

**Syntax:** %PRIORITY = n

**Details:**

- n is the priority and is defined as follows:
  - 1 to 89 : lower than motion, higher than user interface
  - 90 to 99 : lower than user interface
- The lower the value, the higher the task priority.
- The default priority is 50. Refer to [Section 17.4, TASK SCHEDULING](#) for more information on how the specified priority is converted into the system priority.
- The priority can be set during task execution by the SET\_TSK\_ATTR built-in routine.

**Example:** Usually an error handling task pends on an error and when an error occurs, it processes the error recovery as soon as possible. In this case, the error handling task might need to have a higher priority than other tasks, so n should be less than 50.

```
%PRIORITY = 49
```

**Figure A.16.20 %PRIORITY Translator Directive**

## A.16.21 PROG\_BACKUP Built-In Procedure

---

**Purpose:** Saves the specified program and all called programs from execution memory to a storage device. If the called programs call other programs they will be saved recursively. You can specify that any associated program variables be saved.

**Syntax:** PROG\_BACKUP (file\_spec, prog\_type, max\_size, write\_prot, status)

Input/Output Parameters:

- [in] file\_spec : STRING
- [in] prog\_type : INTEGER
- [in] max\_size : INTEGER
- [in] write\_prot : BOOLEAN
- [out] status : INTEGER

%ENVIRONMENT Group : CORE

### Details:

- file\_spec specifies the device and program to save. If a file type is specified, it is ignored.
- prog\_type specifies the type of programs to be saved. The valid types are:
  - PBR\_VRTYPE : VR - programs which contain variables
  - PBR\_MNTYPE : JB, PR, MR, TP
  - PBR\_JBTYP : JB - job programs only
  - PBR\_PRTYPE : PR - process programs only
  - PBR\_MRTYPE : MR - macro programs only
  - PBR\_PCTYPE : VR - saves VR files not PC files
  - PBR\_ALLTYPE : all programs VR, JB, PR, MR, TP
  - PBR\_NVRTYPE : all programs except VR
  - PBR\_NMRTYPE : JB, PR, TP (all TPs except Macros)
- max\_size specifies the maximum size of disk space in kilobytes required to backup the programs. If not enough memory is available on the storage device, no programs will be backed up and status will equal **2002, FILE-002 Device is Full**.
- If the required disk space to backup the programs exceeds max\_size the backup will continue. The backup might still fail if there is not enough space to save all the programs. The return status will equal **2002**, will exist. To prevent this case be sure that max\_size is large enough to prevent this error.
- write\_prot, if TRUE, specifies that write protected programs should be saved. If FALSE, specifies that write protected programs should not be saved.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

- The system will stay in the loop and handle as many programs as it can even when it gets an error. For example, if there is one missing program out of eight, the remaining seven programs are saved. In this case the error Program does not exist is returned to the user in status. An error is posted in this case with the program name in the error for each program. The cause code is whatever is returned from the save routine.
- If a subdirectory is specified on the storage device, it will be created if it does not already exist. All programs will be saved into the subdirectory.
- If a file already exists but the no changes have occurred, the file is not overwritten.
- If a file already exists but the program has been changed, it will be overwritten and no error is returned.
- A KAREL or teach pendant program of the same name with variables must exist in memory as a called program or else the system will not save the VR.
- The PROG\_BACKUP, PROG\_CLEAR and PROG\_RESTORE built-ins consider all references to programs except for macros. This includes any programs referenced in the following statements: CALL, RUN, ERROR\_PROG, RESUME\_PROG, and MONITOR.

**Example:** The following example saves ANS00003 with the appropriate extension to GMX\_211 subdirectory on FR: device. It will save all programs that are called recursively by ANS00003 regardless of program type. It will not save KAREL variables. It will fail if there is less than 200k of free space on the FR: device.

```
VAR
    status: INTEGER
BEGIN
    PROG_BACKUP ('FR:\GMX_211\ANS00003', PBR_NVRTYPE, 200, TRUE,
status)
```

The following example saves ANS00003 with the appropriate file extension to GMX\_211 subdirectory on FR: device. It will save JB, PR, MR, or TP programs that are called recursively by ANS00003. It will not save write-protected programs. It will fail if there is less than 100k of free space on the FR: device.

```
VAR
    status: INTEGER
BEGIN
    PROG_BACKUP ('FR:\GMX_211\ANS00003', PBR_MNTYPE, 100, FALSE,
status)
```

The following example saves MAIN to MC: device with the appropriate file extension. It will save all programs and variables that are called recursively by MAIN. It will fail if there is less than 300k of free space on the MC: device.

```
VAR
    status: INTEGER
BEGIN
    PROG_BACKUP ('MC:\MAIN', PBR_ALLTYPE, 300, TRUE, status)
```

## A.16.22 PROG\_CLEAR Built-In Procedure

**Purpose:** Clear the specified program and all called programs from execution memory. If the called programs call other programs they will be cleared recursively. You can specify that any associated program variables also be cleared. Variables which are referenced from other programs will not be cleared.

**Syntax:** PROG\_CLEAR (prog\_name, prog\_type, status)

Input/Output Parameters:

[in] prog\_name : STRING

[in] prog\_type : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CORE

#### Details:

- prog\_name specifies the root program which is to be cleared.
- prog\_type specifies the type of programs to be cleared. The valid types are:
  - PBR\_VRTYPE : VR - programs which contain variables
  - PBR\_MNTYPE : JB, PR, MR, TP
  - PBR\_JBTYP : JB - job programs only
  - PBR\_PRTYP : PR - process programs only
  - PBR\_MRTYP : MR - macro programs only
  - PBR\_PCTYP : VR - saves VR files not PC files
  - PBR\_ALLTYP : all programs VR, JB, PR, MR, TP
  - PBR\_NVRTYP : all programs except VR
  - PBR\_NMRTYP : JB, PR, TP (all TPs except Macros)
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The system will stay in the loop and clear as many programs as it can. If one of the called programs is missing, this is not an error. However, if that missing program calls other programs those other programs will not be found and will not be cleared. Errors are posted with the program name in the error for each program which is not cleared. The cause code is whatever is returned from the clear routine.
- Clearing of any VR data is subject to being referenced by another program. This error will be ignored for any variable clear operation.
- If a programs which is identified for clearing is the selected program it will not be cleared. The error Program is in use is returned in this case. As a countermeasure the user must use the SELECT\_TPE() built-in to select a program which is not in the clear set.
- The PROG\_BACKUP, PROG\_CLEAR and PROG\_RESTORE built-ins consider all references to programs except for macros. This includes any programs referenced in the following statements: CALL, RUN, ERROR\_PROG, RESUME\_PROG, and MONITOR.

**Example:** The following example clears ANS00003.TP from memory. It will clear all programs that are called recursively by ANS00003 regardless of program type and clear them from memory. It will not clear write-protected programs. It will not clear any KAREL variables.

```
VAR
  status: INTEGER
BEGIN
  PROG_CLEAR('ANS00003.TP', PBR_NVRTYPE, status)
```

The following example clears ANS00003.TP program from memory. It will clear only JB, PR, MR, and TP programs that are called recursively by ANS00003. It will not clear write-protected programs.

```
VAR
  status: INTEGER
```

```
BEGIN
    PROG_CLEAR('ANS00003.TP', PBR_MNTYPE, status)
```

## A.16.23 PROG\_LIST Built-In Procedure

**Purpose:** Returns a list of program names.

**Syntax:** PROG\_LIST(prog\_name, prog\_type, n\_skip, format, ary\_name, n\_progs <,f\_index>)

Input/Output Parameters:

[in] prog\_name : STRING  
[in] prog\_type : INTEGER  
[in] n\_skip : INTEGER  
[in] format : INTEGER  
[out] ary\_name : ARRAY OF STRING  
[out] n\_progs : INTEGER  
[out] status : INTEGER  
[in,out] f\_index : INTEGER  
%ENVIRONMENT Group : BYNAM

**Details:**

- prog\_name specifies the name of the program(s) to be returned in ary\_name. prog\_name may use the wildcard (\*) character, to indicate that all programs matching the prog\_type should be returned in ary\_name.
- prog\_type specifies the type of programs to be retrieved. The valid types are:
  - 1 : VR - programs which contain only variables
  - 2 : JB, PR, MR, TP3 :JB - job programs only
  - 4 : PR - process programs only
  - 5 : MR - macro programs only
  - 6 : PC - KAREL programs only
  - 7 : all programs VR, JB, PR, MR, TP, PC
  - 8 : all programs except VR
- n\_skip: This parameter can only be used when using \* for the program name and 7 for the program type. Otherwise, use the f\_index parameter for multiple calls when more programs exist than the declared length of ary\_name. Set n\_skip to 0 the first time you use PROG\_LIST. If ary\_name is completely filled with program names, copy the array to another ARRAY of STRINGS and execute PROG\_LIST again with n\_skip equal to n\_skip + n\_progs . The call to PROG\_LIST will then skip the programs found in the previous passes and locate only the remaining programs..
- format specifies the format of the program name. The following values are valid for format:
  - 1 : program name only, no blanks
  - 2 : 'program name program type'
  - total length = 15 or 39 characters
- prog\_name = 12 or 36 characters followed by a space
- prog\_type = 2 characters
- ary\_name is an ARRAY of STRING used to store the program names.

- n\_progs is the number of variables stored in the *ary\_name* .
- status will return zero if successful.
- f\_index is an optional parameter for fast indexing. If you specify prog\_name as a complex wildcard (anything other than the straight \*), then you should use this parameter. The first call to PROG\_LIST set f\_index and n\_skip both to zero. f\_index will then be used internally to quickly find the next prog\_name. DO NOT change f\_index once a listing for a particular prog\_name has begun.

**See Also:** [Section A.22.29, VAR\\_LIST Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.16.24 PROG\_RESTORE Built-In Procedure

**Purpose:** Restores (loads) the specified program and all called programs into execution memory. If the called programs call other programs they will be loaded recursively. Any associated program variables will also be loaded if the VR files exist.

**Syntax:** PROG\_RESTORE (file\_spec, status)

Input/Output Parameters:

[in] file\_spec : STRING

[out] status : INTEGER

%ENVIRONMENT Group : CORE

**Details:**

- file\_spec specifies the storage device and program to restore. If a file type is specified, it is ignored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The system will stay in the loop and handles as many programs as it can even when it gets an error. For example, if there is one missing program out of eight, the remaining seven programs are loaded. In this case the error File does not exist is returned to the user in status. An error is posted in this case with the program name in the error for each program. The cause code is whatever is returned from the load routine.
- If a subdirectory is specified, the called programs are loaded from that subdirectory. Any extra files in that subdirectory will not automatically be loaded.
- If the program already exists, it will not be restored and no error is returned.
- A KAREL or TP program of the same name must already exist in memory as a called program or else the system will not load the VR.
- VR types will be restored even if the variables already exist. That is the system will overwrite any existing variable values with the values saved in the VR file.
- ASCII programs (.LS) cannot be restored.
- If not enough memory is available, then an error is returned and the restore is incomplete.
- The PROG\_BACKUP, PROG\_CLEAR and PROG\_RESTORE built-ins consider all references to programs except for macros. This includes any programs referenced in the following statements: CALL, RUN, ERROR\_PROG, RESUME\_PROG, and MONITOR.

**Example:** The following example restores ANS00003.TP from GMX\_211 subdirectory on FR: device. It will restore all programs that are called recursively by ANS00003 regardless of program type. It will restore VR files if they are in the restore directory.

```
VAR
    status: INTEGER
BEGIN
    PROG_RESTORE ('FR:\GMX_211\ANS00003.TP', status)
```

The following example restores ANS00003.TP from GMX\_211 subdirectory on FR: device. It will restore only TP programs that are called recursively by ANS00003.

```
VAR
    status: INTEGER
BEGIN
    PROG_RESTORE ('FR:\GMX_211\ANS00003', status)
```

The following example restores MAIN from MC: device by finding its file type. It will restore all programs and variables that are called recursively by MAIN.

```
VAR
    status: INTEGER
BEGIN
    PROG_RESTORE ('MC:\MAIN', status)
```

## A.16.25 PROGRAM Statement

**Purpose:** Identifies the program name in a KAREL source program

**Syntax:** PROGRAM prog\_name

where:

prog\_name : a valid KAREL identifier

**Details:**

- It must be the first statement (other than comments) in a program.
- The identifier used to name a program cannot be used in the program for any other purpose, such as to identify a variable or constant.
- prog\_name must also appear in the END statement that marks the end of the executable section of the program.
- The program name can be used to call the program as a procedure routine from within a program in the same way routine names are used to call procedure routines.

**Example:** Refer to [Appendix B, KAREL EXAMPLE PROGRAMS](#) for more detailed examples of how to use the PROGRAM statement.

## A.16.26 PULSE Action

---

**Purpose:** Pulses a digital output port for a specified number of milliseconds

**Syntax:** PULSE DOUT[port\_no] FOR time\_in\_ms

where:

port\_no : an INTEGER variable or literal

time\_in\_ms : an INTEGER

**Details:**

- port\_no must be a valid digital output port number.
- time\_in\_ms specifies the duration of the pulse in milliseconds.
- If time\_in\_ms duration is zero, no pulse will occur. Otherwise, the period is rounded up to the next multiple of 8 milliseconds.
- A pulse always turns on the port at the start of the pulse and turns off the port at the end of the pulse.
- If the port is normally on, negative pulses can be accomplished by setting the port to reversed polarity, or by executing the following sequence:

```
DOUT[n] = FALSE
DELAY x
DOUT[n] = TRUE
```

- NOWAIT is not allowed in a PULSE action.
- If the program is paused while a pulse is in progress, the pulse will end at the correct time.
- If the program is aborted while a pulse is in progress, the port stays in whatever state it was in when the abort occurred.
- If time\_in\_ms is negative or greater than 86,400,000 (24 hours), the program is aborted with an error.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.16.27 PULSE Statement

---

**Purpose:** Pulses a digital output port for a specified number of milliseconds.

**Syntax:** PULSE DOUT[port\_no] FOR time\_in\_ms <NOWAIT>

where:

port\_no : an INTEGER variable or literal

time\_in\_ms : an INTEGER

**Details:**

- port\_no must be a valid digital output port number.
- time\_in\_ms specifies the duration of the pulse in milliseconds.
- If time\_in\_ms duration is zero, no pulse will occur. Otherwise, the period is rounded up to the next multiple of 8 milliseconds.

The actual duration of the pulse will be from zero to 8 milliseconds less than the rounded value.

For example, if 100 is specified, it is rounded up to 104 (the next multiple of 8) milliseconds. The actual duration will be from 96 to 104 milliseconds.

- A pulse always turns on the port at the start of the pulse and turns off the port at the end of the pulse.
- If the port is normally on, negative pulses can be accomplished by setting the port to reversed polarity, or by executing the following sequence:

```
DOUT[n] = FALSE
DELAY x
DOUT[n] = TRUE
```

- If NOWAIT is specified in a PULSE statement, the next KAREL statement will be executed concurrently with the pulse.
- If NOWAIT is not specified in a PULSE statement, the next KAREL statement will not be executed until the pulse is completed.

See Also: [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** In the following example a digital output is pulsed, followed by the pulsing of a second digital output. Because NOWAIT is specified, DOUT[start\_air] will be executed before DOUT[5] is completed.

```
PULSE DOUT[5] FOR (seconds * 1000) NOWAIT
PULSE DOUT[start_air] FOR 50 NOWAIT
```

**Figure A.16.27 PULSE Statement**

## A.16.28 PURGE CONDITION Statement

**Purpose:** Deletes the definition of a condition handler from the system

**Syntax:** PURGE CONDITION[cond\_hand\_no]

where:

cond\_hand\_no : an INTEGER expression

**Details:**

- The statement has no effect if there is no condition handler defined with the specified number.
- The PURGE CONDITION statement is used only to purge global condition handlers.
- The PURGE CONDITION statement will purge enabled conditions.
- If a condition handler with the specified number was previously defined, it must be purged before it is replaced with a new one.

See Also: [ENABLE CONDITION Statement](#) [Chapter 6, CONDITION HANDLERS](#) , [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** In the following example, if the BOOLEAN variable ignore\_cond is TRUE, the global condition handler, CONDITION[1], will be purged using the PURGE statement; otherwise CONDITION[1] is enabled.

```
IF ignore_cond THEN
    PURGE CONDITION[1]
ELSE
```

```
ENABLE CONDITION[1]
ENDIF
```

**Figure A.16.28 PURGE CONDITION Statement**

## A.16.29 PURGE\_DEV Built-In Procedure

---

**Purpose:** Purges the specified memory file device by freeing any used blocks that are no longer needed

**Syntax:** PURGE\_DEV (device, status)

Input/Output Parameters:

[in] device : STRING

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- device specifies the memory file device to purge. device should be set to 'FR:' for FROM disk, 'RD:' for RAM disk, or 'MF:' for both disks.
- The purge operation is only necessary when the device does not have enough memory to perform an operation. The 'FR:' device will return 85001 if the FROM disk is full. The 'RD:' device will return 85020 if the RAM disk is full.
- The purge operation will erase file blocks that were previously used, but no longer needed. These are called garbage blocks. The FROM disk may contain many garbage blocks if files are deleted or overwritten. The RAM disk does not normally contain garbage blocks, but they can occur when power is removed during a file copy.
- The VOL\_SPACE built-in can be used to determine the number of garbage blocks on the FROM disk. Hardware limitations may reduce the number of blocks actually freed.
- The device must be mounted and no files can be open on the device or an error will be returned.
- status explains the status of the attempted operation. If not equal to 0 then an error occurred. 85023 is returned if no errors occurred, but no blocks were purged.

**Example:** Refer to [Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#) for a more detailed program example.

## A.16.30 PUSH\_KEY\_RD Built-In Procedure

---

**Purpose:** Suspend key input from a keyboard device

**Syntax:** PUSH\_KEY\_RD(key\_dev\_name, key\_mask, pop\_index, status)

Input/Output Parameters:

[in] key\_dev\_name : STRING

[in] key\_mask : INTEGER

[out] pop\_index : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Suspends all read requests on the specified keyboard device that uses (either as accept\_mask or term\_mask) any of the specified key classes.
- If there are no read requests active, a null set of inputs is recorded as suspended. This is not an error.
- key\_dev\_name must be one of the keyboard devices already defined:
  - 'TPKB' : Teach Pendant Keyboard Device
  - 'CRKB' : CRT Keyboard Device
- key\_mask is a bit-wise mask indicating the classes of characters that will be suspended. This should be an OR of the constants defined in the include file klevkmsk.kl.
  - kc\_display : Displayable keys
  - kc\_func\_key : Function keys
  - kc\_keypad : Keypad and Edit keys
  - kc\_enter\_key : Enter and Return keys
  - kc\_delete : Delete and Backspace keys
  - kc\_lr\_arw : Left and Right Arrow keys
  - kc\_ud\_arw : Up and Down Arrow keys
  - kc\_other : Other keys (such as Prev)
- pop\_id is returned and should be used in a call to POP\_KEY\_RD to re-activate the read requests.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.16.9, POP\\_KEY\\_RD Built-In Procedure](#)

**Example:** Refer to the [Section A.18.4, READ\\_KB Built-In Procedure](#) for an example.

## A.17 - Q - KAREL LANGUAGE DESCRIPTION

### A.17.1 QUEUE\_ATTACH Built-in Procedure

**Purpose:** Attaches the files to given message type and sets message priority (available in v830P/15 and later).

**Syntax:** QUEUE\_ATTACH(msg\_type, attachments, priority, status)

Input/Output parameters:

[in] msg\_type : STRING

[in] attachments : ARRAY OF STRING

[in] priority : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

%ENVIRONMENT Group : ZDTYPES

**Details:**

- msg\_type is a predefined message type. This tells which messages these attachments are for.

- attachments is an array of attachment files. Attachment files are optional. These files are read and deleted when the message is sent. Therefore, they may remain in the queue for quite some time. Unique file names should be used to avoid overwriting of the file if message is not sent before subsequent send. Attachment files using the MD: device are not deleted. They will always contain the most current information when they are sent.
- priority is a flag (0 or 1) that enables message to be sent now or at the next update.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** *ZDT Operator (Combined User and Data Collector Guide) (B-84264EN)* for more details.

**Example:**

Following program sends user data to ZDT for immediate message processing.

```

PROGRAM ZDTKTEST
%ENVIRONMENT zdttypes

VAR
    alarm      : ALARM_MSG_T
    status     : integer
    attachs   : ARRAY[2] OF STRING[15];
    entry      : INTEGER
    prog_name : string[80]
    priority   : INTEGER

BEGIN
    if(uninit(prog_name)) THEN
        prog_name = '*zdtdata*'
    ENDIF

    alarm.err_no = 430011
    alarm.err_text = 'ZDT 001 TEST Error'
    alarm.cause_text = 'This is cause text'
    alarm.alert = 0

    --Set Attachments
    attachs[1] = 'MD:SUMMARY.DG'
    attachs[2] = 'MD:ERRALL.LS'

    --Set Priority
    priority = 1
    status = 0
    QUEUE_ATTACH('ALARM_MSG_P', attachs, priority, status)
    IF status <> 0 THEN
        write ('QUEUE_ATTACH status = ', status, CR)
        PAUSE
    ENDIF

    --Send Message
    SET_VAR(entry, prog_name, 'ALARM_MSG_P', alarm, status)
    IF status <> 0 THEN
        write ('SET_VAR1 status = ', status, CR)
        PAUSE
    ENDIF
END ZDTKTEST

```

## A.17.2 QUEUE\_TYPE Data Type

**Purpose:** Defines the data type for use in QUEUE built-in routines

**Syntax:** QUEUE\_TYPE = STRUCTURE

n\_entries : INTEGER

sequence\_no : INTEGER

head : INTEGER

tail : INTEGER

ENDSTRUCTURE

**Details:**

- queue\_type is used to initialize and maintain queue data for the QUEUE built-in routines. **Do not change this data; it is used internally.**

**See Also:** [Section A.1.16, APPEND\\_QUEUE Built-In Procedure](#), [Section A.4.14, DELETE\\_QUEUE Built-In Procedure](#), [Section A.9.15, INSERT\\_QUEUE Built-In Procedure](#), [Section A.3.48, COPY\\_QUEUE Built-In Procedure](#), [Section A.7.16, GET\\_QUEUE Built-In Procedure](#), [Section A.9.11, INIT\\_QUEUE Built-In Procedure](#), [Section A.13.2, MODIFY\\_QUEUE Built-In Procedure](#)

## A.18 - R - KAREL LANGUAGE DESCRIPTION

### A.18.1 READ Statement

**Purpose:** Reads data from a serial I/O device or file.

**Syntax:** READ <file\_var> (data\_item {,data\_item})

where:

file\_var : a FILE variable

data\_item : a variable identifier and its optional format specifiers or the reserved word CR

**Details:**

- If file\_var is not specified in a READ statement the default TPDISPLAY is used. %CRTDEVICE directive will change the default to INPUT.
- If file\_var is specified, it must be one of the input devices (INPUT, CRTPROMPT, TPDISPLAY, TPPROMPT) or a variable that was set in the OPEN FILE statement.
- If file\_var attribute was set with the UF option, data is transmitted into the specified variables in binary form. Otherwise, data is transmitted as ASCII text.
- data\_item can be a system variable that has RW access or a user-defined variable.
- When the READ statement is executed, data is read beginning with the next nonblank input character and ending with the last character before the next blank, end of line, or end of file for all input types except STRING.
- If data\_item is of type ARRAY, a subscript must be provided.
- If data\_item is of type PATH, you can specify that the entire path be read, a specific node be READ ([n]), or a range of nodes be READ ([n .. m]).

- Optional format specifiers can be used to control the amount of data read for each data\_item. The effect of format specifiers depends on the data type of the item being read and on whether the data is in text (ASCII) or binary (unformatted) form.
- The reserved word CR, which can be used as a data item, specifies that any remaining data in the current input line is to be ignored. The next data item will be read from the start of the next input line.
- If reading from a file and any errors occur during input, the variable being read and all subsequent variables up to CR in the data list are set uninitialized.
- If file\_var is a window device and any errors occur during input, an error message is displayed indicating the bad data item and you are prompted to enter a replacement for the invalid data item and to reenter all subsequent items.
- Use IO\_STATUS(file\_var) to determine if the read operation was successful.

**NOTE**

READ CR should never be used in unformatted mode.

**See Also:** [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#), for more information on the READ format specifiers, [Section A.9.19, IO\\_STATUS Built-In Function](#), [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.18.2 READ\_DICT Built-In Procedure

**Purpose:** Reads information from a dictionary.

**Syntax:** READ\_DICT(dict\_name, element\_no, ksta, first\_line, last\_line, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] element\_no : INTEGER

[out] ksta : ARRAY OF STRING

[in] first\_line : INTEGER

[out] last\_line : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- dict\_name specifies the name of the dictionary from which to read.
- element\_no specifies the element number to read. This element number is designated with a \$ in the dictionary file.

- ksta is a KAREL STRING ARRAY used to store the information being read from the dictionary text file.
- If ksta is too small to store all the data, then the data is truncated and status is set to **33008**, Dictionary Element Truncated.
- first\_line indicates the array element of ksta, at which to begin storing the information.
- last\_line returns a value indicating the last element used in the ksta array.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred reading the element from the dictionary file.
- &new\_line is the only reserved attribute code that can be read from dictionary text files using READ\_DICT. The READ\_DICT built-in ignores all other reserved attribute codes.

**See Also:** [Section A.1.10, ADD\\_DICT Built-In Procedure](#) , [Section A.23.6, WRITE\\_DICT Built-In Procedure](#) , [Section A.18.9, REMOVE\\_DICT Built-In Procedure](#) . Refer to the program example for the [Section A.4.22, DISCTRL\\_LIST Built-In Procedure](#) , [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.18.3 READ\_DICT\_V Built-In-Procedure

**Purpose:** Reads information from a dictionary with formatted variables.

**Syntax:** READ\_DICT\_V(dict\_name, element\_no, value\_array, ksta, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] element\_no : INTEGER

[in] value\_array : ARRAY OF STRING

[out] ksta : ARRAY OF STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

### Details:

- dict\_name specifies the name of the dictionary from which to read.
- element\_no specifies the element number to read. This number is designated with a \$ in the dictionary file.
- value\_array is an array of variable names that corresponds to each formatted data item in the dictionary text. Each variable name can be specified as ' [prog\_name] var\_name '.
  - [prog\_name] specifies the name of the program that contains the specified variable. If not specified, then the current program being executed is used.
  - var\_name must refer to a static variable.
  - var\_name may contain node numbers, field names, and/or subscripts.
- ksta is a KAREL STRING ARRAY used to store the information that is being read from the dictionary text file.
- If ksta is too small to store all the data, then the data is truncated and status is set to **33008**, Dictionary Element Truncated.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred reading the element from the dictionary file.

- &new\_line is the only reserved attribute code that can be read from dictionary text files using READ\_DICT\_V. The READ\_DICT\_V built-in ignores all other reserved attribute codes.

**See Also:** [Section A.23.6, WRITE\\_DICT Built-In Procedure](#), [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** In the following example, TPTASKEG.TX contains dictionary text information which will display a system variable. This information is the first element in the dictionary. Element numbers start at 0. util\_prog uses READ\_DICT\_V to read in the text and display it on the teach pendant.

```
-----
TPTASKEG.TX
-----
$ "Maximum number of tasks = %d"
-----
UTILITY PROGRAM:
-----
PROGRAM util_prog
%ENVIRONMENT uif
VAR
  ksta: ARRAY[1] OF STRING[40]
  status: INTEGER
  value_array: ARRAY[1] OF STRING[30]
BEGIN
  value_array[1] = '[*system*].$scr.$maxnumtask'
  ADD_DICT('TPTASKEG', 'TASK', dp_default, dp_open, status)
  READ_DICT_V('TASK', 0, value_array, ksta, status)
  WRITE(ksta[i], cr)
END util_prog
```

**Figure A.18.3 READ\_DICT\_V Built-In Procedure**

## A.18.4 READ\_KB Built-In Procedure

**Purpose:** Read from a keyboard device and wait for completion.

**Syntax:** READ\_KB(file\_var, buffer, buffer\_size, accept\_mask, term\_mask, time\_out, init\_data, n\_chars\_got, term\_char, status)

Input/Output Parameters:

[in] file\_var : FILE  
[in] buffer : STRING  
[in] buffer\_size : INTEGER  
[in] accept\_mask : INTEGER  
[in] time\_out : INTEGER  
[in] term\_mask : INTEGER  
[in] init\_data : STRING  
[out] n\_chars\_got : INTEGER  
[out] term\_char : INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Causes data from specified classes of characters to be stored in a user-supplied buffer until a termination condition is met or the buffer is full. Returns to the caller when the read is terminated.
- If you use READ\_KB for the CRT/KB, you will get raw CRT characters returned. To get teach pendant equivalent key codes, you must perform the following function:

```
tp_key = $CRT_KEY_TBL[crt_key + 1]
```

This mapping allows you to use common software between the CRT/KB and teach pendant devices.

- READ\_KB and some other utilities use a variable in your KAREL program called *device\_stat* to establish the association between the KAREL program and user interface display. For example, if you have a task [MAINUIF] which calls READ\_KB, the variable which is used to make the association is [MAINUIF]*device\_stat*. If you do not set *device\_stat*, then you can only read characters in single screen mode, or in the left pane.
- *device\_stat* must be set to the paneID in which your application is running. For the standard single mode/monochrome pendant, *device\_stat*=1. To interact in the right pane, set *device\_stat* = 2. To interact in the lower right pane, set *device\_stat* = 3. External Internet Explorer connections use panes 4-9. For the CRT/KB, set *device\_stat* = 255.
- [MAINUIF]*device\_stat* must be set to the correct paneID before you open the keyboard file that is associated with READ\_KB. The paneID for the iPendant can be either 1, 2 or 3.
- The application running in paneID 1 is stored in \$TP\_CURSCRN. paneID 2 is stored in \$UI\_CURSCRN[1], or in general \$UI\_CURSCRN[*device\_stat*-1]. The CRT applicaton uses \$CT\_CURSCRN.
- file\_var must be open to a keyboard-device. If file\_var is also associated with a window, the characters are echoed to the window.
- The characters are stored in buffer, up to a maximum of buffer\_size or the size of the string, whichever is smaller.
- accept\_mask is a bit-wise mask indicating the classes of characters that will be accepted as input. This should be an OR of the constants defined in the include file klevkmsk.kl.
  - kc\_display : Displayable keys
  - kc\_func\_key : Function keys
  - kc\_keypad : Key-pad and Edit keys
  - kc\_enter\_key : Enter and Return keys
  - kc\_delete : Delete and Backspace keys
  - kc\_lr\_arw : Left and Right Arrow keys
  - kc\_ud\_arw : Up and Down Arrow keys
  - kc\_other : Other keys (such as Prev)
- It is reasonable for accept\_mask to be zero; this means that no characters are accepted as input. This is used when waiting for a single key that will be returned as the term\_char. In this case, buffer\_size would be zero.
- If accept\_mask includes displayable characters, the following characters, if accepted, have the following meanings:
  - Delete characters - If the cursor is not in the first position of the field, the character to the left of the cursor is deleted.
  - Left and right arrows - Position the cursor one character to the left or right from its present position, assuming it is not already in the first or last position already.
  - Up and down arrows - Fetch input previously entered in reads to the same file.

- term\_mask is a bit-wise mask indicating conditions which will terminate the request. This should be an OR of the constants defined in the include file klevkmsk.kl.
  - kc\_display : Displayable keys
  - kc\_func\_key : Function keys
  - kc\_keypad : Key-pad and Edit keys
  - kc\_enter\_key : Enter and Return keys
  - kc\_delete : Delete and Backspace keys
  - kc\_lr\_arw : Left and Right Arrow keys
  - kc\_ud\_arw : Up and Down Arrow keys
  - kc\_other : Other keys (such as Prev)
- time\_out specifies the time, in milliseconds, after which the input operation will be automatically canceled. A value of -1 implies no timeout.
- init\_data\_p points to a string which is displayed as the initial value of the input field. This must not be longer than buffer\_size.
- n\_chars\_got is set to the number of characters in the input buffer when the read is terminated.
- term\_char receives a code indicating the character or other condition that terminated the form. The codes for key terminating conditions are defined in the include file klevkeys.kl. Keys normally returned are pre-defined constants as follows:
  - ky\_up\_arw
  - ky\_dn\_arw
  - ky\_rt\_arw
  - ky\_lf\_arw
  - ky\_enter
  - ky\_prev
  - ky\_f1
  - ky\_f2
  - ky\_f3
  - ky\_f4
  - ky\_f5
  - ky\_next
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

The following example suspends any teach pendant reads, uses READ\_KB to read a single key, and then resumes any suspended reads.

```
PROGRAM readkb
%NOLOCKGROUP
%ENVIRONMENT flbt
%ENVIRONMENT uif
%INCLUDE FR:eklevkmsk
VAR
  file_var: FILE
  key: INTEGER
  n_chars_got: INTEGER
  pop_index: INTEGER
  status: INTEGER
  str: STRING[1]
```

```

BEGIN
    -- Suspend any outstanding TP Keyboard reads
    PUSH_KEY_RD('TPKB', 255, pop_index, status)
    IF (status = 0) THEN
        WRITE (CR, 'pop_index is ', pop_index)
    ELSE
        WRITE (CR, 'PUSH_KEY_RD status is ', status)
    ENDIF
    -- Open a file to TP Keyboard with PASALL and FIELD attributes
    -- and NOECHO
    SET_FILE_ATR(file_var, ATR_PASSALL)
    SET_FILE_ATR(file_var, ATR_FIELD)
    OPEN FILE file_var ('RW', 'KB:TPKB')
    -- Read a single key from the TP Keyboard
    READ_KB(file_var, str, 1, 0, kc_display+kc_func_key+kc_keypad+
        kc_enter_key+kc_lr_arw+kc_ud_arw+kc_other, 0, '',
        n_chars_got, key, status)
    IF (status = 0) THEN
        WRITE (CR, 'key is ', key, ', n_chars_got = ', n_chars_got)
    ELSE
        WRITE (CR, 'READ_KB status is ', status)
    ENDIF
    CLOSE FILE file_var
    -- Resume any outstanding TP Keyboard reads
    POP_KEY_RD('TPKB', pop_index, status)
    IF (status <> 0) THEN
        WRITE (CR, 'POP_KEY_RD status is ', status)
    ENDIF
END readkb

```

**Figure A.18.4 READ\_KB Built-In Procedure**

## A.18.5 REAL Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as a REAL data type with a numeric value that includes a decimal point and a fractional part, or numbers expressed in scientific notation.

**Syntax:** REAL

**Details:**

- REAL variables and expressions can have values in the range of -3.4028236E+38 through -1.175494E-38, 0.0, and from +1.175494E-38 through +3.4028236E+38, with approximately seven decimal digits of significance. Otherwise, the program will be aborted with the "Real overflow" error.
- The decimal point is mandatory when defining a REAL constant or literal (except when using scientific notation). The decimal point is not mandatory when defining a REAL variable as long as it was declared as REAL.
- Scientific notation is allowed and governed by the following rules:
  - The decimal point is shifted to the left so that only one digit remains in the INTEGER part.
  - The fractional part is followed by the letter E (upper or lower case) and ±an INTEGER. This part specifies the magnitude of the REAL number. For example, 123.5 is expressed as 1.235E2.
  - The fractional part and the decimal point can be omitted. For example, 100.0 can be expressed as 1.000E2, as 1.E2, or 1E2.
- All REAL variables with magnitudes between -1.175494E- 38 and +1.175494E-38 are treated as 0.0.

- Only REAL or INTEGER expressions can be assigned to REAL variables, returned from REAL function routines, or passed as arguments to REAL parameters.
- If an INTEGER expression is used in any of these instances, it is treated as a REAL value. If an INTEGER variable is used as an argument to a REAL parameter, it is always passed by value, not by reference.
- Valid REAL operators are (refer to [Table A.18.5](#)):
  - Arithmetic operators (+, +, \*, /)
  - Relational operators (>, >=, =, <, <=)

**Table A.18.5 Valid and Invalid REAL operators**

VALID	INVALID	REASON
1.5	15	Decimal point is required (15 is an INTEGER not a REAL)
1.	.	Must include an INTEGER or a fractional part
+2500.450	+2,500.450	Commas not allowed
1.25E-4	1.25E -4	Spaces not allowed

**Example:** Refer to the following sections for detailed program examples:

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.18.6 Relational Condition

**Purpose:** Used to test the relationship between two operands.

**Syntax:** *variable* <[subscript]> *rel\_op expression*

where:

*variable* : a static INTEGER or REAL variable or a BOOLEAN port array element

*subscript* : an INTEGER expression (only used with port arrays)

*rel\_op* : a relational operator

*expression* : a static variable, constant, or EVAL clause

### Details:

- Relational conditions are state conditions, meaning the relationship is tested during every scan.
- The following relational operators can be used:

= : equal

<> : not equal

< : less than

=< : less than or equal

> : greater than

>= : greater than or equal

- Both operands must be of the same data type and can only be of type INTEGER, REAL, or BOOLEAN. INTEGER values can be used where REAL values are required, and will be treated as REAL values.
- *variable* can be any of the port array signals, a user-defined static variable, or a system variable that can be read by a KAREL program.
- *expression* can be a user-defined static variable, a system variable that can be read by a KAREL program, any constant, or an EVAL clause.
- Variables used in relational conditions must be initialized before the condition handler is enabled.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING](#) (DOUT\_EX.KL) for a detailed program example.

## A.18.7 RELAX HAND Statement

**Purpose:** Turns off open signal for a tool controlled by one signal or turns off both open and close signals for a tool controlled by a pair of signals.

**Syntax:** RELAX HAND hand\_num

where:

hand\_num : an INTEGER expression

**Details:**

- The actual effect of the statement depends on how the HAND signals are set up. Refer to [Chapter 16, INPUT/OUTPUT SYSTEM](#).
- hand\_num must be a value in the range 1-2. Otherwise, the program is aborted with an error.
- The statement has no effect if the value of hand\_num is in range but the hand is not connected.
- If the value of hand\_num is in range but the HAND signal represented by that value has not been assigned, the program is aborted with an error.

**See Also:** [Chapter 16, INPUT/OUTPUT SYSTEM](#), [Appendix E, SYNTAX DIAGRAMS](#), for more syntax information

**Example:** In the following example, the robot hand, specified by gripper, is relaxed using the RELAX HAND statement. The robot then moves to the POSITION pstart before closing the hand.

```
PROGRAM p_release
%NOPAUSE=TPENABLE
%ENVIRONMENT uif
BEGIN
  RELAX HAND gripper
  SET_POS_REG(1, pstart, status) — Put position in PR[1]
  move_to_prl — Call TP program to move to PR[1]
  CLOSE HAND gripper
END p_release
```

**Figure A.18.7 RELAX HAND Statement**

## A.18.8 RELEASE Statement

---

**Purpose:** Releases all motion control of the robot arm and auxiliary or extended axes from the KAREL program so that they can be controlled by the teach pendant while a KAREL program is running.

**Syntax:** RELEASE

**Details:**

- Motion stopped prior to execution of the RELEASE statement can only be resumed after the execution of the next ATTACH statement.
- If motion is initiated from the program while in a released state, the program is aborted with the following error, MCTRL Denied because Released.
- If RELEASE is executed while motion is in progress or in a HOLD condition, the program is aborted with the following error, Detach request failed.
- All motion control from all KAREL tasks will be released.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.18.9 REMOVE\_DICT Built-In Procedure

---

**Purpose:** Removes the specified dictionary from the specified language or from all existing languages.

**Syntax:** REMOVE\_DICT(dict\_name, lang\_name, status)

Input/Output Parameters:

[in] dict\_name : STRING

[in] lang\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : UIF

**Details:**

- dict\_name specifies the name of the dictionary to remove.
- lang\_name specifies which language the dictionary should be removed from. One of the following pre-defined constants should be used:

- dp\_default
- dp\_english
- dp\_japanese
- dp\_french
- dp\_german
- dp\_spanish

If lang\_name is "", it will be removed from all languages in which it exists.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred removing the dictionary file.

**See Also:** [Section A.1.10, ADD\\_DICT Built-In Procedure](#), [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.18.10 RENAME\_FILE Built-In Procedure

**Purpose:** Renames the specified file name.

**Syntax:** RENAME\_FILE(old\_file, new\_file, nowait\_sw, status)

Input/Output Parameters:

[in] old\_file : STRING

[in] new\_file : STRING

[in] nowait\_sw : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : FDEV

**Details:**

- old\_file specifies the device, name, and type of the file to rename.
- new\_file specifies the name and type of the file to rename to.
- If nowait\_sw is TRUE, execution of the program continues while the command is executing. If it is FALSE, the program stops, including condition handlers, until the operation has completed. If you have time critical condition handlers in your program, put them in another program that executes as a separate task.

**NOTE**

nowait\_sw is not available in this release and should be set to FALSE.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.46, COPY\\_FILE Built-In Procedure](#) [Section A.4.12, DELETE\\_FILE Built-In Procedure](#)

## A.18.11 RENAME\_VAR Built-In Procedure

**Purpose:** Renames a specified variable in a specified program to a new variable name.

**Syntax:** RENAME\_VAR(prog\_nam, old\_nam, new\_nam, status)

Input/Output Parameters:

[in] prog\_nam : STRING

[in] old\_nam : STRING

[in] new\_nam : STRING

[out] status : INTEGER

%ENVIRONMENT Group : MEMO

**Details:**

- prog\_nam is the name of the program that contains the variable to be renamed.
- old\_nam is the current name of the variable.
- new\_nam is the new name of the variable.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.53, CREATE\\_VAR Built-In Procedure](#) , [Section A.19.38, SET\\_VAR Built-In Procedure](#)

## A.18.12 RENAME\_VARS Built-In Procedure

---

**Purpose:** Renames all of the variables in a specified program to a new program name.

**Syntax:** RENAME\_VARS(old\_nam, new\_nam, status)

Input/Output Parameters:

[in] old\_nam : STRING

[in] new\_nam : STRING

[out] status : INTEGER

%ENVIRONMENT Group : MEMO

**Details:**

- old\_nam is the current name of the program.
- new\_nam is the new name of the program.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.53, CREATE\\_VAR Built-In Procedure](#) , [Section A.18.11, RENAME\\_VAR Built-In Procedure](#)

## A.18.13 REPEAT ... UNTIL Statement

---

**Purpose:** Repeats statement(s) until a BOOLEAN expression evaluates to TRUE.

**Syntax:** REPEAT

{ statement }

UNTIL boolean\_exp

where:

statement : a valid KAREL executable statement

boolean\_exp : a BOOLEAN expression

**Details:**

- boolean\_exp is evaluated after execution of the statements in the body of the REPEAT loop to determine if the statements should be executed again.
- statement continues to be executed and the boolean\_exp is evaluated until it equals TRUE.
- statement will always be executed at least once.

**⚠ CAUTION**

Make sure your REPEAT statement contains a boolean flag that is modified by some condition, and an UNTIL statement that terminates the loop. If it does not, your program could loop infinitely.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES \(CHG\\_DATA.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.18.14 RESET Built-In Procedure

**Purpose:** Resets the controller.

**Syntax:** RESET(successful)

Input/Output Parameters:

[out] successful : BOOLEAN

%ENVIRONMENT Group : MOTN

**Details:**

- successful will be TRUE even if conditions exist which prevent resetting the controller.
- To determine whether the reset operation was successful, delay 1 second and check OPOUT [3] (FAULT LED). If this is FALSE, the reset operation was successful.
- The statement following the RESET built-in is not executed until the reset fails or has completed. The status display on the CRT or teach pendant will indicate PAUSED during the reset.
- The controller appears to be in a PAUSED state while a reset is in progress but, during this time, PAUSE condition handlers will not be triggered.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.18.15 RESUME Action

**Purpose:** Restarts the last stopped motion issued by the task.

**Syntax:** RESUME <GROUP[n{,n}]>

**Details:**

- A motion set is a group of motions issued but not yet terminated when a STOP statement or action is issued.
- If there are no stopped motion sets, no motion will result from the RESUME.
- If more than one motion set has been stopped, RESUME restarts the most recently stopped, unresumed motion set. Subsequent RESUMES will start the others in last-in-first-out sequence.
- The motions contained in a stopped motion set are resumed in the same order in which they were originally issued.
- If a motion is in progress when the RESUME action is issued, any resumed motion(s) occur after the current motion is completed.
- If the group clause is not present, all groups for which the task has control (when the condition is defined) will be resumed.
- If the motion that is stopped, resumed, canceled, or held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are stopped, resumed, canceled, or held.
- Motion cannot be resumed for a different task.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.18.16 RESUME Statement

**Purpose:** Restarts the last stopped motion issued by the task.

**Syntax:** RESUME <GROUP[n{,n}]>

- A motion set is a group of motions issued but not yet terminated when a STOP statement or action is issued.
- If there are no stopped motion sets, no motion will result from the RESUME.
- If more than one motion set has been stopped, RESUME restarts the most recently stopped, unresumed motion set. Subsequent RESUMES will start the others in last-in-first-out sequence.
- Those motions in a stopped motion set are resumed in the same order in which they were originally issued.
- If a motion is in progress when the RESUME statement is issued, any resumed motion(s) occur after the current motion is completed.
- If the group clause is not present, all groups for which the task has control will be resumed.
- If the motion that is stopped, resumed, canceled, or held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are stopped, resumed, canceled, or held.
- Motion cannot be resumed for a different task.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** In the following example, motion is stopped if DIN[1] is ON. It is resumed after F1 is pressed.

```
CONDITION[1] :
  WHEN DIN[1] = ON DO
    STOP
  ENDCONDITION
  move_to_pr1 — Call TP program to move to PR[1]
  IF DIN[1] THEN
    WRITE(' Motion stopped')
```

```
WRITE(CR, 'Motion and the program will resume')
WRITE(CR, '    when F1 of teach pendant is pressed')
WAIT FOR TPIN[129]
RESUME
ENDIF
```

Figure A.18.16 RESUME Statement

## A.18.17 RETURN Statement

---

**Purpose:** Returns control from a routine/program to the calling routine/program, optionally returning a result.

**Syntax:** RETURN <(value)>

**Details:**

- value is required when returning from functions, but is not permitted when returning from procedures. The data type of value must be the same as the type used in the function declaration.
- If a main program executes a RETURN statement, execution is terminated and cannot be resumed. All motions in progress will be completed normally.
- If no RETURN is specified, the END statement serves as the return.
- If a function routine returns with the END statement instead of a RETURN statement, the program is aborted with the **12321** error, END STMT of a func rtn.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.18.18 RGET\_PORTCMT Built-In Routine

---

**Purpose:** In V7.70 and later, to allow a KAREL program to determine a comment that is set for a specified logical port of a remote host.

**Syntax:** RGET\_PORTCMT (host\_port, port\_type, port\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] port\_type : INTEGER  
[in] port\_no : INTEGER  
[out] comment\_str : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of remote robot controller.

- `port_type` specifies the code for the type of port whose comment is returned. Codes are defined in `kliotyps.kl`.
- `port_no` specifies the port number whose comment is being returned.
- `comment_str` is returned with the comment for the specified port. This should be declared as a STRING with a length of at least 16 characters. Only 16 characters are returned even if the server's comment is longer.

**Example:** The following program gets the comment of DO [10] of host RC1

```
PROGRAM RGTPCMT
%COMMENT='RC1 DO[10] CMT'
%NOLOCKGROUP %ENVIRONMENT RPCC
%INCLUDE kliotyps
VAR
    status : INTEGER
    comment : STRING[16]
BEGIN
    RGET_PORTCMT('RC1', io_dout, 10, comment, status)
END RGTPCMT
```

## A.18.19 RGET\_PORTSIM Built-In Routine

---

**Purpose:** In V7.70 and later, to get port simulation status from the remote controller.

**Syntax:** `RGET_PORTSIM(host_port, port_type, port_no, simulated, status)`

Input/Output Parameters

[in] `host_port` : STRING  
[in] `port_type` : INTEGER  
[in] `port_no` : INTEGER  
[out] `simulated` : BOOLEAN  
[out] `status` : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- `host_port` is an IP address or host name of the remote robot controller.
- `port_type` specifies the code for the type of port whose simulation status is returned. Codes are defined in `kliotyps.kl`.
- `port_no` specifies the port number whose simulation status is being returned.
- `simulated` returns TRUE if the port is being simulated, FALSE otherwise.
- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program gets the simulation status of DI [8] of host RC1.

```
ROGRAM RGTPSIM
%COMMENT = 'RC1 DIN[8] SIM'
%NOLOCKGROUP
%ENVIRONMENT RPCC
%include kliotyps
VAR
    status : INTEGER
```

```
    sim_stat: BOOLEAN
BEGIN
    RGET_PORTSIM('RC1', io_din, 8, sim_stat, status)
END RGTPSIM
```

## A.18.20 RGET\_PORTVAL Built-In Routine

**Purpose:** In V7.70 and later, to allow a KAREL program to determine the current value of a specified logical port of a remote host.

**Syntax:** RGET\_PORTVAL (host\_port, port\_type, port\_no, port\_value, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] port\_type : INTEGER  
[in] port\_no :INTEGER  
[out] port\_value :INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of a remote robot controller.
- port\_type specifies the code of a port whose value is being returned. Codes are defined in kliotyps.kl.
- port\_no specifies the port number whose value is returned.
- value is returned with the current value (status) of the specified port. For BOOLEAN port types (DIN for example), this will be 0 = OFF, or 1 = ON.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program gets the value of RO [3] of host RC1.

```
PROGRAM RGTPVAL
%COMMENT='GET RC1 RO[3]'
%NOLOCKGROUP
%ENVIRONMENT RPCC
%include kliotyps
VAR
    status  : INTEGER
    port_val: INTEGER
BEGIN
    RGET_PORTVAL('RC1', io_rdo, 3, port_val, status)
END RGTPVAL
```

## A.18.21 RGET\_PREGCMT Built-In Routine

**Purpose:** In V7.70 and later, to retrieve a comment of a position register of remote host.

**Syntax:** RGET\_PREGCMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
 [in] register\_no : INTEGER  
 [out] comment\_str : STRING  
 [out] status : INTEGER  
 %ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of a remote robot controller.
- register\_no specifies from which position register to retrieve the comment. The comment of the given position register is returned in parameter comment\_str.
- comment\_str should be declared as a STRING with a length of at least 16 characters.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Following program gets the comment of PR[5] of host RC1.

```
PROGRAM RGTPRCMT
%COMMENT='RC1 PR[5] CMT'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
  status : INTEGER
  comment : STRING[16]
BEGIN
  RGET_PREGCMT('RC1', 5, comment, status)
END RGTPRCMT
```

## A.18.22 RGET\_REG Built-In Routine

**Purpose:** In V7.70 and later, to get an INTEGER or REAL value from the specified register of remote host.

**Syntax:** RGET\_REG(host\_port, register\_no, real\_flag, int\_value, real\_value, status)

Input/Output Parameters:

[in] host\_port : STRING  
 [in] register\_no : BOOLEAN  
 [out] real\_flag : INTEGER  
 [out] int\_value : INTEGER  
 [out] real\_value : REAL  
 [out] status : INTEGER  
 %ENVIRONMENT Group : RPCC

**Details:**

- host\_port is the IP address or host name of remote robot controller.
- register\_no specifies which position register to retrieve the comment from. The comment of the given position register is returned in parameter comment\_str.

- real\_flag is set to TRUE and real\_value is set to the register content if the specified register has a real value. Otherwise, real\_flag is set to FALSE and int\_value is set to the contents of the register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** Following program gets value of R[2] of host RC1.

```
PROGRAM RGTNREG
%COMMENT='GET RC1 R[2]'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status      : INTEGER
    is_real     : BOOLEAN
    int_val     : INTEGER
    real_val    : REAL
BEGIN
    RGET_REG('RC1', 2, is_real, int_val, real_val, status)
END RGTNREG
```

## A.18.23 RGET\_REG\_CMT Built-In Routine

**Purpose:** In V7.70 and later, to get comment from the specified register of a remote host.

**Syntax:** RGET\_REG\_CMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] register\_no : INTEGER  
[out] comment\_str : STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

### Details:

- host\_port is an IP address or host name of a remote robot controller.
- register\_no specifies from which register to retrieve the comment. The comment of the given register is returned in comment\_str.
- comment\_str should be declared as a STRING with a length of at least 16 characters.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program gets the comment of R[2] of host RC1.

```
PROGRAM RGTNRRCMT
%COMMENT='GET RC1 R[2] CMT'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status      : INTEGER
    comment    : STRING[16]
BEGIN
```

```
RGET_REG_CMT('RC1', 2, comment, status)
END RGTNRCMT
```

## A.18.24 RGET\_SREGCMT Built-in Routine

---

**Purpose:** In V7.70 and later, to get comment from the specified string register of a remote host.

**Syntax:** RGET\_SREGCMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

- [in] host\_port : STRING
- [in] register\_no : INTEGER
- [out] comment\_str : STRING
- [out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of a remote robot controller.
- register\_no specifies from which string register to retrieve the comment.
- comment\_str contains the comment of the specified string register.
- comment\_str should be declared as a STRING with a length of at least 16 characters.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program gets comment of SR[20] of host RC1.

```
PROGRAM RGTSRCMT
%NOLOCKGROUP
%COMMENT = 'GET CMT SR[20]'
%ENVIRONMENT RPCC
VAR
    status : INTEGER
    comment : STRING[16]
BEGIN
    RGET_SREGCMT('RC1', 20, comment, status)
END RGTSRCMT
```

## A.18.25 RGET\_STR\_REG Built-In Routine

---

**Purpose:** In V7.70 and later, to get a value from the specified string register of a remote host.

**Syntax:** RGET\_STR\_REG (host\_port, register\_no, value, status)

Input/Output Parameters:

- [in] host\_port : STRING
- [in] register\_no : INTEGER
- [out] value : STRING[254]
- [out] status: INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of a remote robot controller.
- register\_no specifies the string register to get.
- value contains the value of the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program gets value of SR[10] or host RC1.

```
PROGRAM RGTSREG
%NOLOCKGROUP
%ENVIRONMENT RPCC
%COMMENT = 'GET SR[10]'
VAR
    status : INTEGER
    sreg_val : STRING[254]
BEGIN
    RGET_STR_REG('RC1', 10, sreg_val, status)
END RGTSREG
```

## **A.18.26 RMCN\_ALERT Built-In Routine**

**Purpose:** Sends an alert to the phone so that the user is immediately notified without requiring the iRConnect mobile application to be active. (available in v8.20 and later).

**Syntax:** RMCN\_ALERT(alertaddr, subject, message, alerturl, status)

Input/Output parameters:

[in] alertaddr : STRING  
[in] subject : STRING  
[in] message : STRING  
[in] alerturl : STRING  
[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- alertaddr is an SMS gateway E-mail address which arrives as a text message on a phone. It can carry multiple Email addresses separated by semicolon (;). If "", then the alert address shown in the **iRConnect Emails** screen will be used.
- subject is the subject for the alert. If "", then '**iRConnect Alert**' is used.
- message is a short message to show in the text message. If "", then only the link is shown.
- alerturl is the link that shows in the text message. When clicked, it will activate the iRConnect mobile application on the phone. If "", then the default link is shown.

**See Also:** iRConnect chapter in the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for more details.

**Example:** The following program sends an alert to iRConnect for immediate E-mail processing.

```

PROGRAM RCMTEST
%NOLOCKGROUP
VAR
    alertaddr: STRING[80]
    subject: STRING[80]
    message: STRING[80]
    alerturl: STRING[80]
    status: INTEGER
BEGIN
    -- Send alert to iRConnect with default parameters
    RMCN_ALERT('', '', '', '', status)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, ERSEV_WARN)
    ENDIF

    -- Send custom alert to iRConnect
    alertaddr = '2483777000@txt.att.net'
    subject = 'MT_PIK05'
    message = 'Drawers not locked in position'
    alerturl = 'iRConnect: irconnect://alerts'
    RMCN_ALERT(alertaddr, subject, message, alerturl, status)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, ERSEV_WARN)
    ENDIF
END RCMTEST

```

## A.18.27 RMCN\_SEND Built-in Routine

**Purpose:** Sends user defined message as an email via iRConnect (available in v8.10 and v830P/14 or earlier).

**Syntax:** RMCN\_SEND(xml\_file, attachments, priority, status)

Input/Output parameters:

- [in] xml\_file : STRING
- [in] attachments : ARRAY OF STRING
- [in] priority : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : PBCORE

### Details:

- xml\_file is a message file in XML format. This file is read and deleted when the message is queued so the same filename can be reused if RMCN\_SEND is called multiple times.
- attachments is an array of attachment files. Attachment files are optional. These files are read and deleted when the email is sent. Therefore, they may remain in the queue for quite some time. Unique filenames should be used if RMCN\_SEND is called multiple times during a queue period, or you can check if the file has been deleted to know when it was sent. Attachment files using the MD: device are not deleted. They will always contain the most current information when they are sent.
- priority is a flag (0 or 1) that enables email to be sent now or at the next update.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** iRConnect Chapter in the *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for more details.

**Example:** The following program sends user data to iRConnect for immediate E-mail processing.

```
PROGRAM RCMTEST
%NOLOCKGROUP
VAR
    xml_file : STRING[20]
    priority : INTEGER
    status : INTEGER
    attachs : ARRAY[2] OF STRING[20]
    noattach: ARRAY[1] OF STRING[1]
BEGIN
    priority = 1
    xml_file = 'td:pick.xml'
    attachs[1] = 'md:errall.ls'
    attachs[2] = 'td:prod_data.txt'

    -- Send data to iRConnect
    RMCN_SEND(xml_file, attachs, priority, status)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, ERSEV_WARN)
    ENDIF

    -- Send data to iRConnect without attachments
    noattach[1] = ''
    RMCN_SEND('td:pick2.xml', noattach, 0, status)
    IF status <> 0 THEN
        POST_ERR(status, '', 0, ERSEV_WARN)
    ENDIF

END RCMTEST
```

## A.18.28 RNUMREG\_RECV Built-In Routine

**Purpose:** In V7.70 and later, to transfer a server's register to a client's register.

**Syntax:** RNUMREG\_RECV (host\_port, src\_idx, dest\_idx, option, status)

Input/Output parameters:

[in] host\_port : STRING  
[in] src\_idx : INTEGER  
[in] dest\_idx : INTEGER  
[in] option : INTEGER  
[in] status : INTEGER

%ENVIRONMENT Group : RPCC

### Details:

- host\_port is an IP address or host name of a remote robot controller.
- src\_idx specifies an index of a register of server.
- dest\_idx specifies an index of register of client. Acquired data is stored in this register.

- option specifies function of this built-in.

**Table A.18.28 Parameter Values**

<b>Value</b>	<b>Description</b>
0	Value and comment are received.
1	Value is received.
2	Comment is received.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sends value of R[11] to R[2] of host RC1.

```
PROGRAM RNREGRCV
%COMMENT='RC1 R3->12'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    STATUS : INTEGER
BEGIN
    RNUMREG_RECV('RC1', 3, 12, 0, STATUS)
END RNREGRCV
```

## A.18.29 RNUMREG\_SEND Built-In Routine

**Purpose:** In V7.70 and later, to transfer a client's register to a server's register.

**Syntax:** RNUMREG\_SEND (host\_port, dest\_idx, src\_idx, option, status)

Input/Output parameters:

[in] host\_port : STRING  
[in] dest\_idx : INTEGER  
[in] src\_idx : INTEGER  
[in] option : INTEGER  
[in] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is an IP address or host name of a remote robot controller.
- dest\_idx specifies an index of register of server.
- src\_idx specifies an index of register of client. Data of this register is sent to client's server.
- option specifies function of this built-in.

**Table A.18.29 Parameter Values**

<b>Value</b>	<b>Description</b>
0	Value and comment is set.
1	Value is set.
2	Comment is set.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sends the value of R[11] to R[2] of host RC1.

```
PROGRAM RNREGSND
%COMMENT='R11->RC1 R1'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    STATUS : INTEGER
BEGIN
    RNUMREG_SEND('RC1', 2, 11, 1, STATUS)
END RNREGSND
```

## A.18.30 ROUND Built-In Function

**Purpose:** Returns the INTEGER value closest to the specified REAL argument.

**Syntax:** ROUND(x)

Function Return Type : INTEGER

Input/Output Parameters:

[in] x : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the INTEGER value closest to the REAL value x, as demonstrated by the following rules:
  - If  $x \geq 0$ , let n be a positive INTEGER such that  $n \leq x \leq n + 1$
  - If  $x \geq n + 0.5$ , then  $n + 1$  is returned; otherwise, n is returned.
  - If  $x \leq 0$ , let n be a negative INTEGER such that  $n \geq x \geq n - 1$
  - If  $x \leq n - 0.5$ , then  $n - 1$  is returned; otherwise, n is returned.
- x must be in the range of -2147483648 to +2147483646. Otherwise, the program will be aborted with an error.

**See Also:** [Section A.20.5, TRUNC Built-In Function](#)

**Example:** Refer to [Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_.EX.KL\)](#), for a detailed program example.

## A.18.31 ROUTINE Statement

**Purpose:** Specifies a routine name, with parameters and types, and a returned value data type for function routines.

**Syntax:** ROUTINE name < param\_list > <: return\_type >

where:

name : a valid KAREL identifier

param\_list : described below

return\_type : any data type that can be returned by a function, that is, any type except FILE, PATH, and vision types

#### Details:

- name specifies the routine name.
- param\_list is of the form ( name\_group { ; name\_group } )
  - name\_group is of the form param\_name : param\_type
  - param\_name is a parameter which can be used within the routine body as a variable of data type param\_type.
  - If a param\_type or return\_type is an ARRAY, the size is excluded. If the param\_type is a STRING, the string length is excluded.
- When the routine body follows the ROUTINE statement, the names in param\_list are used to associate arguments passed in with references to parameters within the routine body.
- When a routine is from another program, the names in the parameter list are of no significance but must be present in order to specify the number and data types of parameters.
- If the ROUTINE statement contains a return\_type, the routine is a function routine and returns a value. Otherwise, it is a procedure routine.
- The ROUTINE statement must be followed by a routine body or a FROM clause.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.4, STANDARD ROUTINES \(ROUT\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.18.32 RPREG\_RECV Built-In Routine

**Purpose:** In V7.70 and later, to transfer position register of specified group of server to position register of specified group of client.

**Syntax:** RPOSREG\_RECV (host\_port, src\_idx, src\_grp, dest\_idx, dest\_grp, option, status)

Input/Output Parameters:

[in] host\_port : STRING

[in] src\_idx : INTEGER

[in] src\_grp : INTEGER

[in] dest\_idx : INTEGER

[in] dest\_grp : INTEGER  
 [in] option : INTEGER  
 [in] status : INTEGER  
 %ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- src\_idx specifies index of position register or server.
- src\_grp specifies group number of server. Position data of specified group is transferred.
- dest\_idx specifies index of position register of client. The specified position register stores acquired data.
- dest\_grp specifies group number of client. Position data of specified group of PR [dest\_idx] is changed to received data.
- option explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Table A.18.32 Parameter Values**

Value	Description
0	Position data and comment are got.
1	Position data is got.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following example program performs the following transfer. Comment is transferred, too.

From: Group 1 of PR [5] of host RC1

To: Group 2 of PR [6]

```
PROGRAM RPREGRCV
%COMMENT='RC1 PR5G1->PR6G2'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    STATUS : INTEGER
BEGIN
    RPOSREG_RECV('RC1', 5, 1, 6, 2, 0, STATUS)
END RPREGRCV
```

## A.18.33 RPREG\_SEND Built-in Routine

**Purpose:** In V7.70 and later, to transfer position register of specified group of client to position register of specified group of server.

**Syntax:** RPOSREG\_SEND (host\_port, dest\_idx, dest\_grp, src\_idx, src\_grp, option, status)

Input/Output Parameters:

[in] host\_port : STRING  
 [in] dest\_idx : INTEGER  
 [in] dest\_grp : INTEGER  
 [in] src\_idx : INTEGER

[in] src\_grp : INTEGER  
 [in] option : INTEGER  
 [in] status : INTEGER  
 %ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- dest\_idx specifies index of position register of server.
- dest\_grp specifies group number of server. Position data of specified group is changed by sent data from client.
- src\_idx specifies index of position register of client. Position data of src\_grp of PR [src\_idx] is sent to server.
- src\_grp specifies group number of client.
- option specifies function of this built-in.

**Table A.18.33 Parameter Values**

VALUE	DESCRIPTION
0	Position data and comment are set.
1	Position data is set.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following example program performs the following transfer. Comment is transferred, too.

From: Group 2 of PR [7] of client

To: Group1 of PR [8] of host RC1

```
PROGRAM RPREGSND
%COMMENT='PR7G2->RC1 PR8G1'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
  STATUS : INTEGER
BEGIN
  RPOSREG_SEND('RC1', 8, 1, 7, 2, 0, STATUS)
END RPREGSND
```

## A.18.34 RSET\_INT\_REG Built-in Routine

**Purpose:** In V7.70 and later, to store an in INTEGER value in the specified register of remote host.

**Syntax:** RSET\_INT\_REG (host\_port, register\_no, int\_value, status)

Input/Output Parameters:

[in] host\_port : STRING  
 [in] register\_no : BOOLEAN  
 [in] int\_value : INTEGER  
 [out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies the register into which int\_value will be stored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets 100 to R[20] of host RC1.

```
PROGRAM RSTNRI
%COMMENT = 'RC R[20]=100'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status : INTEGER
BEGIN
    RSET_INT_REG('RC1', 20, 100, status)
END RSTNRI
```

## A.18.35 RSET\_PORTCMT Built-in Routine

**Purpose:** In V7.70 and later, to allow a KAREL program to set comment of specified logical port of remote host.

**Syntax:** RSET\_PORTCMT (host\_port, port\_type, port\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] port\_type : INTEGER  
[in] port\_no : INTEGER  
[in] comment\_str : INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- port\_type specifies the code of port whose comment is being set. Codes are defined in kliotyps.kl.
- port\_no specifies the port number whose comment is being set.
- comment\_str is a string whose value is the comment for the specified port. This must not be over 16 characters long.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets the comment to DO[10] of host RC1.

```
PROGRAM RSTPCMT
%COMMENT ='RC1 DO[10] cmt'
%NOLOCKGROUP
%ENVIRONMENT RPCC
%INCLUDE kliotyps
VAR
    status : INTEGER
```

```
BEGIN
    RSET_PORTCMT('RC1', io_dout, 10, 'RC1 DOUT[10]', status)
END RSTPCMT
```

## A.18.36 RSET\_PORTSIM Built-in Routine

---

**Purpose:** In V7.70 and later, to set port simulated on remote host.

**Syntax:** RSET\_PORTSIM(host\_port, port\_type, port\_no, value, status)

Input/Output Parameters:

- [in] host\_port : STRING
- [in] port\_type : INTEGER
- [in] port\_no : INTEGER
- [in] value : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- port\_type specifies the code for the type of port to be simulated. Codes are defined in kliotyps.kl
- port\_no specifies port number to be simulated.
- value specifies the initial value to set.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets GO[3] of RC1 simulated and set its initial simulated value to 5.

```
PROGRAM RSTPSIM
%COMMENT ='RC1 SIM GO[3], 5'
%NOLOCKGROUP
%ENVIRONMENT RPCC
%INCLUDE kliotyps
VAR
    status : INTEGER
BEGIN
    RSET_PORTSIM('RC1', io_gpout, 3, 5, status)
END RSTPSIM
```

## A.18.37 RSET\_PORTVAL Built-in Routine

---

**Purpose:** In V7.70 and later, to allow KAREL program to set a specified output (or simulated input) for a specified logical port.

**Syntax:** RSET\_PORTVAL(host\_port, port\_type, port\_no, port\_value, status)

Input/Output Parameters:

- [in] host\_port : STRING

[in] port\_type : INTEGER  
[in] port\_no : INTEGER  
[in] port\_value : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- port\_type specifies the code of port whose value is being set. Codes are defined in kliotyps.kl
- port\_no specifies the port number whose value is being set.
- value indicates the value to be assigned to a specified port. If the port\_type is BOOLEAN (DOUT for example), this should be 0 = OFF, or 1 = ON. This field can be used to set input ports if the port is simulated.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets DO [3] of host RC1 to ON.

```
PROGRAM RSTPVAL
%COMMENT ='RC1 DO[3]=ON'
%NOLOCKGROUP
%ENVIRONMENT RPCC
%INCLUDE kliotyps
VAR
    status : INTEGER
BEGIN
    RSET_PORTVAL('RC1', io_dout, 3, 1, status)
END RSTPVAL
```

## A.18.38 RSET\_PREGCMT Built-in Routine

**Purpose:** In V7.70 and later, to set comment of a position register of remote host.

**Syntax:** RSET\_PREGCMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] register\_no : INTEGER  
[in] comment\_str : STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies position register of remote robot controller.
- comment\_str is comment to be set. If comment\_str exceeds 16 characters, it is truncated.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets the comment 'RC1 PR[5]' to PR[5] of host RC1

```
PROGRAM RSTPRCMT
%COMMENT = 'RC1 PR[5] CMT'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status : INTEGER
BEGIN
    RSET_PREGCMT('RC1', 5, 'RC1 PR[5]', status)
END RSTPRCMT
```

## A.18.39 RSET\_REALREG Built-in Routine

**Purpose:** In V7.70 and later, to store a REAL value in the specified register of remote host.

**Syntax:** RSET\_REALREG (host\_port, register\_no, real\_value, status)

Input/Output Parameters:

- [in] host\_port : STRING
- [in] register\_no : BOOLEAN
- [in] real\_value : REAL
- [out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies the register into which real\_value will be stored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets -12.3 to R[20] of RC1.

```
PROGRAM RSTNRR
%COMMENT = 'RC1 R[20]=-12.3'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status : INTEGER
BEGIN
    RSET_REALREG('RC1', 20, -12.3, status)
END RSTNRR
```

## A.18.40 RSET\_REG\_CMT Built-In Routine

**Purpose:** In V7.70 and later, to set comment of numeric register of remote host.

**Syntax:** RSET\_REG\_CMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] register\_no : INTEGER  
[in] comment\_str : STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies which register to set the comments to.
- comment\_str represents the data which is to be used to set the comment of the given register. If comment\_str exceeds more than 16 characters, the built-in will truncate the string.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets the comment 'RC1 R[15]' to R[15] of host RC1

```
PROGRAM RSTNRCMT
%COMMENT = 'RC1 R[15] CMT'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status : INTEGER
BEGIN
    RSET_REG_CMT('RC1', 15, 'RC1 R[15]', status)
END RSTNRCMT
```

## A.18.41 RSET\_SREGCMT Built-in Routine

**Purpose:** In V7.70 and later, sets the comment for the specified string register of the remote robot controller.

**Syntax:** RSET\_SREGCMT (host\_port, register\_no, comment\_str, status)

Input/Output Parameters:

[in] host\_port : STRING  
[in] register\_no : INTEGER  
[in] comment\_str : STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies string register to set
- comment\_str contains the comment to set to the specified string register. If the comment\_str exceeds more than 16 characters, the built-in will truncate the string
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets comment 'RC1\_STRING\_REG\_3' to SR[3] of host RC1.

```
PROGRAM RSTSRCMT
%COMMENT ='RC1 SR[3] CMT'
%NOLOCKGROUP
%ENVIRONMENT RPCC
VAR
    status : INTEGER
BEGIN
    RSET_SREGCMT('RC1', 3, 'RC1_STRING_REG_3', status)
END RSTSRCMT
```

## A.18.42 RSET\_STR\_REG Built-in Routine

**Purpose:** In V7.70 and later, to set the specified value for the specified string register of the remote host.

**Syntax:** RSET\_STR\_REG (host\_port, register\_no, value, status)

Input/Output Parameters:

- [in] host\_port : STRING
- [in] register\_no : INTEGER
- [out] value : STRING[254]
- [out] status : INTEGER

%ENVIRONMENT Group : RPCC

**Details:**

- host\_port is IP address or host name of remote robot controller.
- register\_no specifies string register to set
- value contains the value to set to the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following program sets SR[4] of host RC1.

```
PROGRAM RSTSREG
%COMMENT ='Set RC1 SR[4] '
%NOLOCKGROUP
%ENVIRONMENT rpcc
CONST
    ERR_SUCCESS = 0
VAR
    status : INTEGER
BEGIN
    RSET_STR_REG('RC1', 4, 'RC1 of SR[4] was set by remote host',
    status)
END RSTSREG
```

## A.18.43 RUN\_TASK Built-In Procedure

**Purpose:** Runs the specified program as a child task.

**Syntax:** RUN\_TASK (task\_name, line\_number, pause\_on\_sft, tp\_motion, lock\_mask, status)

Input/Output Parameters:

[in] task\_name : STRING  
[in] line\_number : INTEGER  
[in] pause\_on\_sft : BOOLEAN  
[in] tp\_motion : BOOLEAN  
[in] lock\_mask : INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- task\_name is the name of the task to be run. This creates a child task. The task that executes this built-in is called the parent task.
- If the task already exists and is paused, it will be continued. A new task is not created.
- line\_number specifies the line from which execution starts. Use 0 to start from the beginning of the program. This is only valid for teach pendant programs.
- If pause\_on\_sft is TRUE, the task is paused when the teach pendant SHIFT key is released.
- If tp\_motion is TRUE, the task can execute motion while the teach pendant is enabled. The TP must be enabled if tp\_motion is TRUE.
- The control of the motion groups specified in lock\_mask will be transferred from parent task to child task, if tp\_motion is TRUE and the teach pendant is enabled. The group numbers must be in the range of 1 to the total number of groups defined on the controller. Bit 1 specifies group 1, bit 2 specifies group 2, and so forth.

**Table A.18.43 Group\_mask Setting**

GROUP	DECIMAL	BIT
Group 1	1	1
Group 2	2	2
Group 3	4	3

To specify multiple groups select the decimal values, shown in [Table A.18.43](#), which correspond to the desired groups. Then connect them together using the OR operator. For example to specify groups 1 and 3, enter "1 OR 4".

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.45, CONT\\_TASK Built-In Procedure](#) , [Section A.16.6, PAUSE\\_TASK Built-In Procedure](#) , [Section A.1.4, ABORT\\_TASK Built-In Procedure](#) , [Chapter 17, MULTI-TASKING](#)

**Example:** Refer to [Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#), for a detailed program example.

## A.19 - S - KAREL LANGUAGE DESCRIPTION

## A.19.1 SAVE Built-In Procedure

---

**Purpose:** Saves the program or variables into the specified file

**Syntax:** `SAVE (prog_nam, file_spec, status)`

Input/Output Parameters:

[in] `prog_nam` : STRING

[in] `file_spec` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : MEMO

**Details:**

- `prog_nam` specifies the program name. If program name is '`*`', all programs or variables of the specified type are saved. `prog_name` must be set to `*SYSTEM*` in order to save all system variables.
- `file_spec` specifies the device, name, and type of the file being saved to. The type also implies whether programs or variables are being saved. The following types are valid:
  - `.TP` : Teach pendant program
  - `.VR` : KAREL variables
  - `.SV` : KAREL system variables
  - `.IO` : I/O configuration data
- If `file_spec` already exists on the specified device, then an error is returned the save does not occur.
- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.3.10, CLEAR Built-In Procedure](#) , [Section A.12.2, LOAD Built-In Procedure](#)

**Example:** Refer to [Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VRS.KL\)](#), for a detailed program example.

## A.19.2 SAVE\_DRAM Built-In Procedure

---

**Purpose:** Saves the RAM variable content to FlashROM.

**Syntax:** `SAVE_DRAM (prog_nam, status)`

Input/Output Parameters:

[in] `prog_nam` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : MEMO

**Details:**

- `prog_nam` specifies the program name. This operation will save the current values of any variables in DRAM to FlashROM for the specified program. At power up these saved values will automatically be loaded into DRAM.
- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.

## A.19.3 SELECT ... ENDSELECT Statement

**Purpose:** Permits execution of one out of a series of statement sequences, depending on the value of an INTEGER expression.

**Syntax:** SELECT case\_val OF

CASE(value{,value}):

{statement}

{ CASE(value{, value}):

{statement} }

<ELSE:

{ statement }>

ENDSELECT

where:

case\_val : an INTEGER expression

value : an INTEGER constant or literal

statement : a valid KAREL executable statement

**Details:**

- case\_val is compared with each of the values following the CASE in each clause. If it is equal to any of these, the statements between the CASE and the next clause are executed.
- Up to 1000 CASE clauses can be used in a SELECT statement.
- If the same INTEGER value is listed in more than one CASE, only the statement sequence following the first matching CASE will be executed.
- If the ELSE clause is used and the expression case\_val does not match any of the values in the CASE clauses, the statements between the keywords ELSE and ENDSELECT are executed.
- If no ELSE clause is used and the expression case\_val does not match any of the values in the CASE clauses, the program is aborted with the No match in CASE error.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

## A.19.4 SELECT\_TPE Built-In Procedure

**Purpose:** Selects the program of the specified name

**Syntax:** SELECT\_TPE(prog\_name, status)

Input/Output Parameters:

[in] prog\_name : STRING

[out] status : INTEGER  
 %ENVIRONMENT Group : TPE

**Details:**

- prog\_name specifies the name of the program to be selected as the teach pendant default. This is the program that is in use by the teach pendant. It is also the program that will be executed if the CYCLE START button is pressed or the teach pendant FWD key is pressed.
- status explains the status of the attempted operation. If it is not equal to 0, then an error has occurred.

**See Also:** [Section A.15.3, OPEN\\_TPE Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.19.5 SEMA\_COUNT Built-In Function

---

**Purpose:** Returns the current value of the specified semaphore

**Syntax:** SEMA\_COUNT (semaphore\_no)

Function Return Type: INTEGER

Input/Output Parameters:

[in] semaphore\_no : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- The value of the semaphore indicated by semaphore\_no is returned.
- This value is incremented by every POST\_SEMA call and SIGNAL\_SEMAPHORE action specifying the same semaphore\_no. It is decremented by every PEND\_SEMA call.
- If SEMA\_COUNT is greater than zero, a PEND\_SEMA call will fall through immediately. If it is -n (minus n), then there are n tasks pending on this semaphore.

**See Also:** [Section A.16.18, POST\\_SEMA Built-In Procedure](#) , [Section A.16.7, PEND\\_SEMA Built-In Procedure](#) , [Section A.3.11, CLEAR\\_SEMA Built-In Procedure](#) , [Chapter 17, MULTI-TASKING](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.19.6 SEMAPHORE Condition

---

**Purpose:** Monitors the value of the specified semaphore

**Syntax:** SEMAPHORE[semaphore\_no]

**Details:**

- semaphore\_no specifies the semaphore number to use.
- semaphore\_no must be in the range of 1 to the number of semaphores defined on the controller.
- When the value of the indicated semaphore is greater than zero, the condition is satisfied (TRUE).

## A.19.7 SEND\_DATAPC Built-In Procedure

**Purpose:** To send an event message and other data to the PC.

**Syntax:** SEND\_DATAPC(event\_no, dat\_buffer, status)

Input/Output Parameters:

[in] event\_no : INTEGER

[in] dat\_buffer : ARRAY OF BYTE

[out] status : INTEGER

%ENVIRONMENT Group : PC

**Details:**

- event\_no – a GEMM event number. Valid values are 0 to 255.
- dat\_buffer – an array of up to 244 bytes. The KAREL built-ins ADD\_BYNAMEPC, ADD\_INTPC, ADD\_REALPC, and ADD\_STRINGPC can be used to format a KAREL byte buffer. The actual data buffer format depends on the needs of the PC. There is no error checking of the dat\_buffer format on the controller.
- status – the status of the attempted operation. If not 0, then an error occurred and the event request was not sent to the PC.

**See Also:** [Section A.1.9, ADD\\_BYNAMEPC Built-In Procedure](#), [Section A.1.11, ADD\\_INTPC Built-In Procedure](#), [Section A.1.12, ADD\\_REALPC Built-In Procedure](#), [Section A.1.13, ADD\\_STRINGPC Built-In Procedure](#)

**Example:** The following example sends event 12 to the PC with a data buffer.

```
PROGRAM TESTDATA
%ENVIRONMENT PC
CONST
    er_abort = 2
VAR
    dat_buffer:    ARRAY[100] OF BYTE
    index:         INTEGER
    status:        INTEGER
BEGIN
    index = 1
    ADD_INTPC(dat_buffer, index, 55, status)
    ADD_REALPC(dat_buffer, index, 123.5, status)
    ADD_STRINGPC(dat_buffer, index, 'YES', status)

    -- send event 12 and data buffer to PC
    SEND_DATAPC(12, dat_buffer, status)
    IF status<>0 THEN
        POST_ERR(status, '', 0, er_abort)
    ENDIF
END testdata
```

**Figure A.19.7 SEND\_DATAPC Built-In Procedure**

## A.19.8 SEND\_EVENTPC Built-In Procedure

**Purpose:** To send an event message to the PC.

**Syntax:** SEND\_EVENTPC(event\_no, status)

Input/Output Parameters:

[in] event\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PC

**Details:**

- event\_no – a GEMM event number. Valid values are 0 through 255.
- status – the status of the attempted operation. If not 0, then an error occurred and the event request was not sent to the PC.

**Example:** The following example sends event 12 to the PC.

```
PROGRAM TESTEVT
%ENVIRONMENT PC
CONST
    er_abort = 2
VAR
    status: INTEGER
BEGIN
    -- send event 12 to PC
    SEND_EVENTPC(12,status)      -- call built-in here
    IF status<>0 THEN
        POST_ERR(status,'',0,er_abort)
    ENDIF
END testevt
```

Figure A.19.8 SEND\_EVENTPC Built-In Procedure

## A.19.9 SET\_ATTR\_PRG Built-In Procedure

**Purpose:** Sets attribute data of the specified teach pendant or KAREL program

**Syntax:** SET\_ATTR\_PRG(program\_name, attr\_number, int\_value, string\_value, status)

Input/Output Parameters:

[in] program\_name : STRING

[in] attr\_number : INTEGER

[in] int\_value : INTEGER

[in] string\_value : STRING

[out] status : INTEGER

%ENVIRONMENT Group : TPE

**Details:**

- program\_name specifies the program to which attribute data is set.
- attr\_number is the attribute whose value is to be set. The following attributes are valid:
  - AT\_PROG\_TYPE : (#) Program type
  - AT\_PROG\_NAME : Program name (String[12])
  - AT\_OWNER : Owner (String[8])
  - AT\_COMMENT : Comment (String[16])
  - AT\_PROG\_SIZE : (#) Size of program
  - AT\_ALLC\_SIZE : (#) Size of allocated memory
  - AT\_NUM\_LINE : (#) Number of lines
  - AT\_CRE\_TIME : (#) Created (loaded) time
  - AT\_MDFY\_TIME : (#) Modified time
  - AT\_SRC\_NAME : Source file ( or original file ) name (String[128])
  - AT\_SRC\_VRSN : Source file version
  - AT\_DEF\_GROUP : Default motion groups (for task attribute)
  - AT\_PROTECT : Protection code; 1 :protection OFF; 2 : protection ON
  - AT\_STORAGE : Storage type:
    - TPSTOR\_CMOS
    - TPSTOR\_SHADOW
    - TPSTOR\_FILE
    - TPSTOR\_SHOD
  - AT\_STK\_SIZE : Stack size (for task attribute)
  - AT\_TASK\_PRI : Task priority (for task attribute)
  - AT\_DURATION : Time slice duration (for task attribute)
  - AT\_BUSY\_OFF : Busy lamp off (for task attribute)
  - AT\_IGNR\_ABRT : Ignore abort request (for task attribute)
  - AT\_IGNR\_PAUS : Ignore pause request (for task attribute)
  - AT\_CONTROL : Control code (for task attribute)
  - (#) : Cannot be set.
- If the attribute data is a number, it is set to int\_value and string\_value is ignored.
- If the attribute data is a string, it is set to string\_value and int\_value is ignored.
- status explains the status of the attempted operation. If it is not equal to 0, then an error has occurred.  
Some of the errors which could occur are:
  - **7073** The program specified in program\_name does not exist
  - **7093** The attribute of a program cannot be set while it is running
  - **17033** attr\_number has an illegal value or cannot be set

## A.19.10 SET\_CURSOR Built-In Procedure

**Purpose:** Set the cursor position in the window

**Syntax:** SET\_CURSOR(file\_var, row, col, status)

Input/Output Parameters:

[in] file\_var : FILE

[in] row : INTEGER

[in] col : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- Sets the current cursor of the specified file that is open to a window so subsequent writes will start in the specified position.
- file\_var must be open to a window.
- A row value of 1 indicates the top row of the window. A col value of 1 indicates the left-most column of the window.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.4.9, DEF\\_WINDOW Built-In Procedure](#)

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

## A.19.11 SET\_EPOS\_REG Built-In Procedure

**Purpose:** Stores an XYZWPREXT value in the specified register

**Syntax:** SET\_EPOS\_REG(register\_no, posn, status <, group\_no>)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] posn : XYZWPREXT

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register in which to store the value.
- The position data is set in XYZWPREXT representation.
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

**See Also:** [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#) , [Section A.19.17, SET\\_JPOS\\_TPE Built-In Procedure](#) , [Section A.7.12, GET\\_POS\\_REG Built-In Function](#) , [Section A.7.3, GET\\_JPOS\\_REG Built-In Function](#)

**Example:** The following example sets the extended position for the specified register.

```
PROGRAM spe
%environment REGOPE
VAR
    cur_pos: XYZWPREXT
    posget: XYZWPREXT
    status: INTEGER
    v_mask, g_mask: INTEGER
    reg_no: INTEGER
BEGIN
    reg_no = 1
    cur_pos = CURPOS(v_mask,g_mask)
    SET_EPOS_REG(reg_no,cur_pos,status)
    posget = GET_POS_REG(reg_no,status)
END spe
```

**Figure A.19.11 SET\_EPOS\_REG Built-In Procedure**

## **A.19.12 SET\_EPOS\_TPE Built-In Procedure**

**Purpose:** Stores an XYZWPREXT value in the specified position in the specified teach pendant program

**Syntax:** SET\_EPOS\_TPE (open\_id, position\_no, posn, status <,group\_no>)

Input/Output Parameters:

[in] open\_id : INTEGER  
[in] position\_no : INTEGER  
[in] posn : XYZWPREXT  
[out] status : INTEGER  
[in] group\_no : INTEGER  
%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the program in which to store the value.
- A motion instruction must already exist that uses the position\_no or the position will not be used by the teach pendant program.
- The position data is set in XYZWPREXT representation with no conversion.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

## A.19.13 SET\_FILE\_ATR Built-In Procedure

---

**Purpose:** Sets the attributes of a file before it is opened

**Syntax:** SET\_FILE\_ATR(file\_id, atr\_type <,atr\_value>)

Input/Output Parameters:

[in] file\_id : FILE

[in] atr\_type : INTEGER expression

[in] atr\_value : INTEGER expression

%ENVIRONMENT Group : PBCORE

**Details:**

- file\_id is the file variable that will be used in the OPEN FILE, WRITE, READ, and/or CLOSE FILE statements.
- atr\_type specifies the attribute type to set. The predefined constants as specified in [Table 7.2.1 \(a\)](#) should be used.
- atr\_value is optional depending on the attribute type being set.

### XML Related

**Purpose:** Sets the attributes file to XML before it is opened

**Syntax:** SET\_FILE\_ATR(xml\_file, ATR\_XML)

Input/Output Parameters:

[in] xml\_file : FILE

[in] ATR\_XML : INTEGER expression

**Details:**

- xml\_file is the file variable that will be used in the OPEN FILE, WRITE, READ, and CLOSE FILE statements.
- ATR\_XML specifies the attribute type to set. The predefined constants as specified in [Table 7.2.1 \(a\)](#) should be used.
- The file must then be opened as a RO file. You cannot do any other XML operations until the file has been opened.

**See Also:** [Section A.19.21, SET\\_PORT\\_ATR Built-In Function](#), [Section 7.2.1, Setting File and Port Attributes](#) and [Section 10.4, FORMATTING XML INPUT](#)

**Example:** Refer to [Section 10.4, FORMATTING XML INPUT](#)

## A.19.14 SET\_FILE\_POS Built-In Procedure

---

**Purpose:** Sets the file position for the next READ or WRITE operation to take place in the specified file to the value of the new specified file position

**Syntax:** SET\_FILE\_POS(file\_id, new\_file\_pos, status)

Input/Output Parameters:

[in] file\_id : FILE  
[in] new\_file\_pos : INTEGER expression  
[out] status : INTEGER variable  
%ENVIRONMENT Group : FLBT

**Details:**

- The file associated with file\_id must be opened and uncompressed on either the FROM or RAM disks. Otherwise, the program is aborted with an error.
- new\_file\_pos must be in the range of -1 to the number of bytes in the file:
  - at\_eof : specifies that the file position is to be set at the end of the file.
  - at\_sof : specifies that the file position is to be set at the start of the file.
  - Any other value causes the file to be set the specified number of bytes from the beginning of the file.
- status is set to 0 if the new\_file\_pos is between -1 and the number of bytes in the file, indicating the file position was successfully set. If not equal to 0, then an error occurred.

**See Also:** [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#)

**Example:** The following example opens the filepos.dt data file, sets the file position from a directory, reads the positions from the file, and stores the positions in the PATH, my\_path.

```
OPEN FILE file_id ('RW', 'filepos.dt')
FOR i = 1 TO PATH_LEN(my_path) DO
    SET_FILE_POS(file_id, pos_dir[i], status)
    IF status = 0 THEN
        READ file_id (temp_pos)
        my_path[i].node_pos = temp_pos
    ENDIF
ENDFOR
```

**Figure A.19.14 SET\_FILE\_POS Built-In Procedure**

## A.19.15 SET\_INT\_REG Built-In Procedure

**Purpose:** Stores an integer value in the specified register

**Syntax:** SET\_INT\_REG(register\_no, int\_value, status)

Input/Output Parameters:

[in] register\_no : INTEGER  
[in] int\_value : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the register into which int\_value will be stored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.19.29, SET\\_REAL\\_REG Built-In Procedure](#)

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## A.19.16 SET\_JPOS\_REG Built-In Procedure

---

**Purpose:** Stores a JOINTPOS value in the specified register

**Syntax:** SET\_JPOS\_REG(register\_no, jpos, status<, group\_no>)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] jpos : JOINTPOS

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register in which to store the position, jpos.
- The position data is set in JOINTPOS representation with no conversion.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

**See Also:** [Section A.7.3, GET\\_JPOS\\_REG Built-In Function](#) , [Section A.7.12, GET\\_POS\\_REG Built-In Function](#) , [Section A.19.26, SET\\_POS\\_REG Built-In Procedure](#) , [Section A.16.14, POS\\_REG\\_TYPE Built-In Procedure](#)

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## A.19.17 SET\_JPOS\_TPE Built-In Procedure

---

**Purpose:** Stores a JOINTPOS value in the specified position in the specified teach pendant program

**Syntax:** SET\_JPOS\_TPE(open\_id, position\_no, posn, status<, group\_no>)

Input/Output Parameters:

[in] open\_id : INTEGER

[in] position\_no : INTEGER

[in] posn : JOINTPOS

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. Before calling this built-in, a program must be opened using the OPEN\_TPE built-in, and have read/write access.
- position\_no specifies the position in the program in which to store the value.
- The position data is set in JOINTPOS representation with no conversion.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

**See Also:** [Section A.7.4, GET\\_JPOS\\_TPE Built-In Function](#) , [Section A.7.13, GET\\_POS\\_TPE Built-In Function](#) , [Section A.19.27, SET\\_POS\\_TPE Built-In Procedure](#) , [Section A.7.14, GET\\_POS\\_TYP Built-In Procedure](#)

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY.TP.KL\)](#), for a detailed program example.

## A.19.18 SET\_LANG Built-In Procedure

**Purpose:** Changes the current language

**Syntax:** SET\_LANG(lang\_name, status)

Input/Output Parameters:

[in] lang\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- lang\_name specifies which language from which the dictionaries should be read/written. Use one of the following pre-defined constants:
  - dp\_default
  - dp\_english
  - dp\_japanese
  - dp\_french
  - dp\_german
  - dp\_spanish
- The read-only system variable \$LANGUAGE indicates which language is currently in use.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred setting the language.
- The error, **33003**, No dict found for language, will be returned if no dictionaries are loaded into the specified language. The KCL command SHOW LANGS can be used to view which languages are created in the system.

**See Also:** [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to [Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#), for a detailed program example.

## A.19.19 SET\_PERCH Built-In Procedure

---

**Purpose:** Sets the perch position and tolerance for a group of axes

**Syntax:** SET\_PERCH(jpos, tolerance, indx)

Input/Output Parameters:

[in] jpos : JOINTPOS

[in] tolerance : ARRAY[6] of REAL

[in] indx : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- The values of *jpos* are converted to radians and stored in the system variable *\$REFPOS1[indx].\$perch\_pos*.
- The tolerance array is converted to degrees and stored in the system variable *\$REFPOS1[indx].\$perchtol*. If the tolerance array is uninitialized, an error is generated.
- *indx* specifies the element number to be set in the *\$REFPOS1* array.
- The group of axes is implied from the specified position, *jpos*. If JOINTPOS is not in group 1, then the system variable *\$REFPOSn* is used where n corresponds to the group number of *jpos* and *indx* must be set to 1.

**See Also:** Your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN) to set up reference positions.

**Example:** In the following example, *\$REFPOS1[2].\$perchpos* and *\$REFPOS1[2].\$perchtol* are set according to *perch\_pos* and *tolerance[i]*.

```

VAR
    perch_pos: JOINTPOS IN GROUP[1]
BEGIN
    FOR i = 1 to 6 DO
        tolerance[i] = 0.01
    ENDFOR
    SET_PERCH (perch_pos, tolerance, 2)
END

```

**Figure A.19.19 SET\_PERCH Built-In Procedure**

## A.19.20 SET\_PORT\_ASG Built-In Procedure

---

**Purpose:** Allows a KAREL program to assign one or more logical ports to specified physical port(s)

**Syntax:** SET\_PORT\_ASG(log\_port\_type, log\_port\_no, rack\_no, slot\_no, phy\_port\_type, phy\_port\_no, n\_ports, status)

Input/Output Parameters:

```
[in] log_port_type : INTEGER
[in] log_port_no : INTEGER
[in] rack_no : INTEGER
[in] slot_no : INTEGER
[in] phy_port_type : INTEGER
[in] phy_port_no : INTEGER
[in] n_ports : INTEGER
[out] status : INTEGER
%ENVIRONMENT Group : IOSETUP
```

**Details:**

- `log_port_type` specifies the code for the type of port to be assigned. Codes are defined in `KLIOTYPS.KL`.
- `log_port_no` specifies the number of the port to be assigned.
- `rack_no` is the rack containing the port module. For process I/O boards, memory-image, and dummy ports, this is zero; for Allen-Bradley and Genius ports, this is 16.
- `slot_no` is the slot containing the port module. For process I/O boards, this is the sequence in the SLC-2 chain. For memory-image and dummy ports, this is zero; for Allen-Bradley and Genius ports, this is 1.
- `phy_port_type` is the type of port to be assigned to. Often this will be the same as `log_port_type`. Exceptions are if `log_port_type` is a group type (`io_gpin` or `io_gpout`) or a port is assigned to memory-image or dummy ports.
- `phy_port_no` is the number of the port to be assigned to. If `log_port_type` is a group, this is the port number for the least-significant bit of the group.
- `n_ports` is the number of physical ports to be assigned to the logical port. If `log_port_type` is a group type, `n_ports` indicates the number of bits in the group. When setting digital I/O, `n_ports` is the number of points you are configuring. In most cases this will be 8, but may be 1 through 8.
- `status` is returned with zero if the parameters are valid. Otherwise, it is returned with an error code. The assignment is invalid if the specified port(s) do not exist or if the assignment of `log_port_type` to `phy_port_type` is not permitted.

For example, GINs cannot be assigned to DOUTs. Neither `log_port_type` nor `phy_port_type` can be a system port type (SOPIN, for example).

**NOTE**

The assignment does not take effect until the next power-up.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.21 SET\_PORT\_ATR Built-In Function

**Purpose:** Sets the attributes of a port

**Syntax:** `SET_PORT_ATR(port_id, atr_type, atr_value)`

Function Return Type: INTEGER

Input/Output Parameters:

[in] port\_id : INTEGER  
 [in] atr\_type : INTEGER  
 [in] atr\_value : INTEGER  
 %ENVIRONMENT Group : FLBT

**Details:**

- port\_id is one of the predefined constants as follows:
  - port\_1
  - port\_2
  - port\_3
  - port\_4
- atr\_type specifies the attribute type to set. One of the following predefined constants should be used:
  - atr\_readahd : Read ahead buffer
  - atr\_baud : Baud rate
  - atr\_parity : Parity
  - atr\_sbts : Stop bits
  - atr\_dbts : Data length
  - atr\_xonoff : XON/XOFF
  - atr\_eol : End of line
  - atr\_modem : Modem line
- atr\_value specifies the value for the attribute type. See [Table A.19.21](#) for acceptable pre-defined attribute types with corresponding values.

**Table A.19.21 Attribute Values**

ATR_TYPE	ATR_VALUE
atr_readahd	any integer, represents multiples of 128 bytes (for example: atr_value=1 means the buffer length is 128 bytes.)
atr_baud	baud_9600 baud_4800 baud_2400 baud_1200
atr_parity	parity_none parity_even parity_odd
atr_sbts	sbts_1 sbts_15 sbts_2

ATTR_TYPE	ATTR_VALUE
atr_dbits	dbits_5 dbits_6 dbits_7 dbits_8
atr_xonoff	xf_not_used xf_used
atr_eol	an ASCII code value, refer to <a href="#">Appendix D, CHARACTER CODES</a>
atr_modem	md_not_used md_use_dsr md_nouse_dsr md_use_dtr md_nouse_dtr md_use_rts md_nouse_rts

- A returned integer is the status of this action to port.

**See Also:** [Section A.19.13, SET\\_FILE\\_ATR Built-In Procedure](#) , [Section 7.2.1, Setting File and Port Attributes](#) for more information

**Example:** Refer to the example for the [Section A.7.6, GET\\_PORT\\_ATR Built-In Function](#) .

## A.19.22 SET\_PORT\_CMT Built-In Procedure

**Purpose:** Allows a KAREL program to set the comment displayed on the teach pendant, for a specified logical port

**Syntax:** SET\_PORT\_CMT(port\_type, port\_no, comment\_str, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] comment\_str : STRING

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

### Details:

- port\_type specifies the code for the type of port whose mode is being set. Codes are defined in KLIOTYPS.KL.
- port\_no specifies the port number whose mode is being set.
- comment\_str is a string whose value is the comment for the specified port. This must not be over 16 characters long.
- status is returned with zero if the parameters are valid and the specified mode can be set for the specified port.

**See Also:** [Section A.19.25, SET\\_PORT\\_VAL Built-In Procedure](#), [Section A.19.23, SET\\_PORT\\_MOD Built-In Procedure](#), [Section A.7.7, GET\\_PORT\\_CMT Built-In Procedure](#), [Section A.7.10, GET\\_PORT\\_VAL Built-In Procedure](#), [Section A.7.8, GET\\_PORT\\_MOD Built-In Procedure](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.23 SET\_PORT\_MOD Built-In Procedure

**Purpose:** Allows a KAREL program to set (or reset) special port modes for a specified logical port

**Syntax:** SET\_PORT\_MOD(port\_type, port\_no, mode\_mask, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] mode\_mask : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port whose mode is being set. Codes are defined in [KLIOTYPS.KL](#).
- port\_no specifies the port number whose mode is being set.
- mode\_mask is a mask specifying which modes are turned on. The following modes are defined:
  - 1 : reverse mode – sense of the port is reversed; if the port is set to TRUE, the physical output is set to FALSE. If the port is set to FALSE, the physical output is set to TRUE. If a physical input is TRUE when the port is read, FALSE is returned. If a physical input is FALSE when the port is read, TRUE is returned.
  - 2 : complementary mode – the logical port is assigned to two physical ports whose values are complementary. In this case, port\_no must be an odd number. If port n is set to TRUE, port n is set to TRUE, and port n + 1 is set to FALSE. If port n is set to FALSE, port n is set to FALSE and port n + 1 is set to TRUE. This is effective only for output

**NOTE**

The mode setting does not take effect until the next power-up.

- status is returned with zero if the parameters are valid and the specified mode can be set for the specified port.

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.24 SET\_PORT\_SIM Built-In Procedure

**Purpose:** Sets port simulated

**Syntax:** SET\_PORT\_SIM(port\_type, port\_no, value, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] value : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port to simulate. Codes are defined in KLIOTYPS.KL.
- port\_no specifies the number of the port to simulate.
- value specifies the initial value to set.
- status is returned with zero if the port is simulated.

**See Also:** [Section A.19.20, SET\\_PORT\\_ASG Built-In Procedure](#), [Section A.7.5, GET\\_PORT\\_ASG Built-in Procedure](#), [Section A.7.9, GET\\_PORT\\_SIM Built-In Procedure](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.25 SET\_PORT\_VAL Built-In Procedure

**Purpose:** Allows a KAREL program to set a specified output (or simulated input) for a specified logical port

**Syntax:** SET\_PORT\_VAL(port\_type, port\_no, value, status)

Input/Output Parameters:

[in] port\_type : INTEGER

[in] port\_no : INTEGER

[in] value : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : IOSETUP

**Details:**

- port\_type specifies the code for the type of port whose mode is being set. Codes are defined in KLIOTYPS.KL.
- port\_no specifies the port number whose mode is being set.
- value indicates the value to be assigned to a specified port. If the port\_type is BOOLEAN (DOUT, for example), this should be 0 = OFF, or 1 = ON. This field can be used to set input ports if the port is simulated.
- status is returned with zero if the parameters are valid and the specified mode can be set for the specified port.

**See Also:** [Section A.19.25, SET\\_PORT\\_VAL Built-In Procedure](#), [Section A.19.23, SET\\_PORT\\_MOD Built-In Procedure](#), [Section A.7.7, GET\\_PORT\\_CMT Built-In Procedure](#), [Section A.7.10, GET\\_PORT\\_VAL Built-In Procedure](#), [Section A.7.8, GET\\_PORT\\_MOD Built-In Procedure](#)

**Example:** The following example sets the value for a specified port.

```

PROGRAM setvalprog
%ENVIRONMENT IOSETUP
%INCLUDE FR:\kliotyps
ROUTINE set_value(port_type: INTEGER;
                  port_no: INTEGER;
                  g_value: BOOLEAN) : INTEGER
VAR
  value: INTEGER
  status: INTEGER
BEGIN
  IF g_value THEN
    value = 1
  ELSE
    value = 0;
  ENDIF
  SET_PORT_VAL (port_type, port_no, value, status)
  RETURN (status)
END set_value
BEGIN
END setvalprog

```

Figure A.19.25 SET\_PORT\_VAL Built-In Procedure

## A.19.26 SET\_POS\_REG Built-In Procedure

**Purpose:** Stores an XYZWPR value in the specified position register

**Syntax:** SET\_POS\_REG(register\_no, posn, status<, group\_no>)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] posn : XYZWPR

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the position register in which to store the value.
- The position data is set in XYZWPR representation with no conversion.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

**Example:** Refer to [Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#), for a detailed program example.

## A.19.27 SET\_POS\_TPE Built-In Procedure

**Purpose:** Stores an XYZWPR value in the specified position in the specified teach pendant program

**Syntax:** SET\_POS\_TPE(open\_id, position\_no, posn, status<, group\_no>)

Input/Output Parameters:

[in] open\_id : INTEGER

[in] position\_no : INTEGER

[in] posn : XYZWPR

[out] status : INTEGER

[in] group\_no : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. Before calling this built-in, a program must be opened using the OPEN\_TPE built-in, and have read/write access.
- position\_no specifies the position in the program in which to store the value.
- A motion instruction must already exist that uses the position\_no or the position will not be used by the teach pendant program.
- The position data is set in XYZWPR representation with no conversion.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.
- If group\_no is omitted, the default group for the program is assumed. Data for other groups is not changed.
- If group\_no is specified, it must be in the range of 1 to the total number of groups defined on the controller.

**Example:** Refer to [Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#), for a detailed program example.

## A.19.28 SET\_PREG\_CMT Built-In-Procedure

**Purpose:** To set the comment information of a KAREL position register based on a given register number and a given comment.

**Syntax:** SET\_PREG\_CMT (register\_no, comment\_string, status)

Input/Output Parameters:

[in] register\_no: INTEGER

[in] comment\_string: STRING

[out] status: INTEGER

%ENVIRONMENT Group: REGOPE

**Details:**

- register\_no specifies the register into which real\_value will be stored.
- comment\_string represents the data which is to be used to set the comment of the given position register.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

## A.19.29 SET\_REAL\_REG Built-In Procedure

---

**Purpose:** Stores a REAL value in the specified register

**Syntax:** SET\_REAL\_REG(register\_no, real\_value, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] real\_value : REAL

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the register into which real\_value will be stored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.19.15, SET\\_INT\\_REG Built-In Procedure](#)

## A.19.30 SET\_REG\_CMT Built-In-Procedure

---

**Purpose:** To set the comment information of a KAREL register based on a given register number and a given comment.

**Syntax:** SET\_REG\_CMT (register\_no, comment\_string, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] comment\_string : STRING

[out] status : INTEGER

%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies which register to retrieve the comments from.
- The comment\_string represents the data which is to be used to set the comment of the given register. If the comment\_string exceeds more than 16 characters, the built-in will truncate the string.

## A.19.31 SET\_SREG\_CMT Built-in Procedure

---

**Purpose:** Sets the comment for the specified string register.

**Syntax:** SET\_SREG\_CMT(register\_no, comment, status)

Input/Output Parameters:

[in] register\_no : INTEGER  
[in] comment : STRING[254]  
[out] status : INTEGER  
%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the string register to get.
- comment contains the comment to set to the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.20, GET\\_STR\\_REG Built-In Procedure](#), [Section A.7.19, GET\\_SREG\\_CMT Built-In Procedure](#), [Section A.7.20, GET\\_STR\\_REG Built-In Procedure](#)

## **A.19.32 SET\_STR\_REG Built-in Procedure**

**Purpose:** Sets the value for the specified string register.

**Syntax:** SET\_STR\_REG(register\_no, value, status)

Input/Output Parameters:

[in] register\_no : INTEGER  
[in] value : STRING[254]  
[out] status : INTEGER  
%ENVIRONMENT Group : REGOPE

**Details:**

- register\_no specifies the string register to get.
- value contains the value to set to the specified string register.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.20, GET\\_STR\\_REG Built-In Procedure](#), [Section A.7.19, GET\\_SREG\\_CMT Built-In Procedure](#), [Section A.19.31, SET\\_SREG\\_CMT Built-in Procedure](#)

## **A.19.33 SET\_TIME Built-In Procedure**

**Purpose:** Set the current time within the KAREL system

**Syntax:** SET\_TIME(i)

Input/Output Parameters:

[in] i : INTEGER  
%ENVIRONMENT Group : TIM

**Details:**

- i holds the INTEGER representation of time within the KAREL system. This value is represented in 32-bit INTEGER format as follows:

**Table A.19.33 32-Bit INTEGER Format of Time**

31–25	24–21	20–16
year	month	day
15–11	10–5	4–0
hour	minute	second

- The contents of the individual fields are as follows:

- DATE:
  - Bits 15-9 — Year since 1980
  - Bits 8-5 — Month (1-12)
  - Bits 4-0 — Day of the month
- TIME:
  - Bits 31-25 — Number of hours (0-23)
  - Bits 24-21 — Number of minutes (0-59)
  - Bits 20-16 — Number of 2-second increments (0-29)

- This value can be determined using the GET\_TIME and CNV\_STR\_TIME built-in procedures.
- If i is 0, the time on the system will not be changed.
- INTEGER values can be compared to determine if one time is more recent than another.

**See Also:** [Section A.3.36, CNV\\_STR\\_TIME Built-In Procedure](#), [Section A.7.21, GET\\_TIME Built-In Procedure](#)

**Example:** The following example converts the STRING variable str\_time, input by the user in DD-MMM-YYY HH:MM:SS format, to the INTEGER representation of time int\_time using the CNV\_STR\_TIME built-in procedure. SET\_TIME is then used to set the time within the KAREL system to the time specified by int\_time.

```
WRITE('Enter the new time : ')
READ(str_time)
CNV_STR_TIME(str_time,int_time)
SET_TIME(int_time)
```

**Figure A.19.33 SET\_TIME Built-In Procedure**

## A.19.34 SET\_TPE\_CMT Built-In Procedure

**Purpose:** Provides the ability for a KAREL program to set the comment associated with a specified position in a teach pendant program.

**Syntax:** SET\_TPE\_CMT(open\_id, pos\_no, comment, status)

Input/Output Parameters:

[in] open\_id : INTEGER

[in] pos\_no : INTEGER

[in] comment : STRING

[out] status : INTEGER

%ENVIRONMENT Group : TPE

**Details:**

- open\_id specifies the open\_id returned from a previous call to OPEN\_TPE. An open mode of TPE\_RWACC must be used in the OPEN\_TPE call.
- pos\_id specifies the number of the position in the teach pendant program to get a comment from. The specified position must have been recorded.
- comment is the comment to be associated with the specified position. A zero length string can be used to ensure that a position has no comment. If the string is over 16 characters, it is truncated and used and a warning error is returned.
- status indicates zero if the operation was successful, otherwise an error code will be displayed.

**See Also:** [Section A.7.22, GET\\_TPE\\_CMT Built-in Procedure](#) , [Section A.15.3, OPEN\\_TPE Built-In Procedure](#)

## **A.19.35 SET\_TRNS\_TPE Built-In Procedure**

**Purpose:** Stores a POSITION value within the specified position in the specified teach pendant program

**Syntax:** SET\_TRNS\_TPE(open\_id, position\_no, posn, status)

Input/Output Parameters:

[in] open\_id : INTEGER  
[in] position\_no : INTEGER  
[in] posn : POSITION  
[out] status : INTEGER  
%ENVIRONMENT Group : PBCORE

**Details:**

- open\_id specifies the opened teach pendant program. A program must be opened before calling this built-in.
- position\_no specifies the position in the program in which to store the value specified by posn. Data for other groups is not changed.
- The position data is set in POSITION representation with no conversion.
- posn is the group number of position\_no.
- status explains the status of the attempted operation. If not equal to 0, then an error has occurred.

## **A.19.36 SET\_TSK\_ATTR Built-In Procedure**

**Purpose:** Set the value of the specified running task attribute

**Syntax:** SET\_TSK\_ATTR(task\_name, attribute, value, status)

Input/Output Parameters:

[in] task\_name : STRING  
[in] attribute : INTEGER  
[in] value : INTEGER

[out] status : INTEGER  
 %ENVIRONMENT Group : PBCORE

**Details:**

- task\_name is the name of the specified running task. A blank task\_name will indicate the calling task.
- attribute is the task attribute whose value is to be set. The following attributes are valid:
  - TSK\_PRIORITY : Priority, see %PRIORITY for value information
  - TSK\_TIMESLIC : Time slice duration, see %TIMESLICE for value information
  - TSK\_NOBUSY : Busy lamp off, see %NOBUSYLAMP
  - TSK\_NOABORT : Ignore abort request
  - Pg\_np\_error : No abort on error
  - Pg\_np\_cmd : No abort on command
  - TSK\_NOPAUSE : Ignore pause request
  - pg\_np\_error : No pause on error
  - pg\_np\_end : No pause on command when TP is enabled
  - pg\_np\_tpenb : No pause
  - TSK\_TRACE : Trace enable
  - TSK\_TRACELEN : Maximum number of lines to store when tracing
  - TSK TPMOTION : TP motion enable, see %TPMOTION for value information
  - TSK\_PAUSESFT : Pause on shift, reverse of %NOPAUSESHFT
- value depends on the task attribute being set.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.24, GET\\_TSK\\_INFO Built-In Procedure](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.19.37 SET\_TSK\_NAME Built-In Procedure

---

**Purpose:** Set the name of the specified task

**Syntax:** SET\_TSK\_NAME(old\_name, new\_name, status)

Input/Output Parameters:

[in] old\_name : STRING  
 [in] new\_name : STRING  
 [out] status : INTEGER  
 %ENVIRONMENT Group : MULTI

**Details:**

- task\_name is the name of the task of interest. A blank task\_name will indicate the calling task.
- new\_name will become the new task name.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.7.1, GET\\_ATTR\\_PRG Built-In Procedure](#)

**Example:** See examples in [Chapter 17, MULTI-TASKING](#)

## A.19.38 SET\_VAR Built-In Procedure

**Purpose:** Allows a KAREL program to set the value of a specified variable

**Syntax:** SET\_VAR(entry, prog\_name, var\_name, value, status)

Input/Output Parameters:

[in,out] entry : INTEGER

[in] prog\_name : STRING

[in] var\_name : STRING

[in] value : Any valid KAREL data type except PATH

[out] status : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- entry returns the entry number in the variable data table of var\_name in the device directory where var\_name is located. **This variable should not be modified.**
- prog\_name specifies the name of the program that contains the specified variable. If prog\_name is ' ', then it defaults to the current task name being executed. prog\_name can also access a system variable on a robot in a ring.
- Use prog\_name of '\*SYSTEM\*' to set a system variable.
- var\_name must refer to a static variable.
- var\_name can contain node numbers, field names, and subscripts.
- If both var\_name and value are ARRAYS, the number of elements copied will equal the size of the smaller of the two arrays.
- If both var\_name and value are STRINGS, the number of characters copied will equal the size of the smaller of the two strings.
- If both var\_name and value are STRUCTUREs of the same type, value will be an exact copy of var\_name.
- var\_name will be set to value.
- If value is uninitialized, the value of var\_name will be set to uninitialized and status will be set to 12311. value must be a static variable within the calling program.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

 **CAUTION**

Using SET\_VAR to modify system variables could cause unexpected results.

- The designated names of all the robots can be found in the system variable \$PH\_MEMBERS[]/. This also include information about the state of the robot. The ring index is the array index for this system variable. KAREL users can write general purpose programs by referring to the names and other information in this system variable rather than explicit names.

**See Also:** [Section A.3.53, CREATE\\_VAR Built-In Procedure](#) , [Section A.18.12, RENAME\\_VARS Built-In Procedure](#) , *Internet Options Setup and Operations Manual (MAROUIN9010171E)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)* for information on accessing system variables on a robot in a ring.

**Example 1:** To access \$TP\_DEFPROG on the MHROB03 robot in a ring, see [Figure A.19.38 \(a\)](#).

```
SET_VAR(entry, '\\MHROB03\*system*', '$TP_DEFPROG', strvar, status)
```

**Figure A.19.38 (a) Accessing \$TP\_DEFPROG on MHROB03**

**Example 2:** In [Figure A.19.38 \(b\)](#), an array [ipgetset]set\_data[x,y] is set on all robots in the ring from all robots in the ring. In this array x is the source robot index and y is the destination robot index:

```
FOR idx = 1 TO $PH_ROSIP.$NUM_MEMBERS DO
  IF idx = $PH_ROSIP.$MY_INDEX THEN
    -- This will work but it this robot so is inefficient
  ELSE
    SELECT $PH_MEMBERS[idx].$STATE OF
      CASE (0) : -- Offline
        sstate = 'Offline'
      CASE (1) : -- Online
        sstate = 'Online'
      CASE (2) : -- Synchronized
        sstate = 'Synch'
        CNV_INT_STR(idx, 1, 10, sidx)
        prog_name = '\\'+$PH_MEMBERS[idx].$NAME+'\ipgetset'
        var_name = 'set_data['+smy_index+', '+sidx+']'
        GET_VAR(entry, prog_name, var_name,
        set_data[$PH_ROSIP.$MY_INDEX,
          idx], status[idx])
        IF status[idx] = 0 THEN
          IF uninit(set_data[$PH_ROSIP.$MY_INDEX, idx]) THEN
            set_data[$PH_ROSIP.$MY_INDEX, idx] = 0
          ELSE
            set_data[$PH_ROSIP.$MY_INDEX, idx] =
        set_data[$PH_ROSIP.$MY_INDEX,
          idx] + 1
          ENDIF
          SET_VAR(entry, prog_name, var_name,
        set_data[$PH_ROSIP.$MY_INDEX, idx],
          status[idx])
        ENDIF
      ENDSELECT
    ENDIF
  ENDFOR
```

**Figure A.19.38 (b) GET\_VAR SET\_VAR Example**

**Example 3:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

**Example 4:** GET\_VAR and SET\_VAR can also be used to set register values.

This will work for the local robot with the program names \*posreg\* and \*numreg\*. For the local robot this has similar functionality to the GET\_POS\_REG, GET\_REG and SET\_REG, SET\_POS\_REG built-ins. The built-ins only work for the local robot. You can access robots in the ring via GET\_VAR and SET\_VAR by using the robot name as part of the program name.

For the case of SET\_VAR on numeric registers the register type will be set to the type of the KAREL variable. In the example below after executing this code numeric register 20 will be an integer and numeric register 21 will be a real.

If a position register is locked and you attempt to set it the error position register locked is returned. See Figure A.19.38 (c).

```
program GETREG
%nolockgroup
VAR
entry: integer
status: integer
int_data: integer
real_data: real
posext_data: xyzwprext
BEGIN
    GET_VAR(entry, '\\mhrob01\\*numreg*', '$NUMREG[10]', int_data,
status)
    IF status <> 0 THEN
        GET_VAR(entry, '\\mhrob01\\*numreg*', '$NUMREG[10]', real_data,
status)
    ENDIF
    GET_VAR(entry, '\\mhrob01\\*posreg*', '$POSREG[1, 10]', 
posext_data, status)
    SET_VAR(entry, '\\mhrob01\\*numreg*', '$NUMREG[20]', int_data,
status)
    SET_VAR(entry, '\\mhrob01\\*numreg*', '$NUMREG[21]', real_data,
status)
    SET_VAR(entry, '\\mhrob01\\*posreg*', '$POSREG[1, 20]', 
posext_data, status)
end GETREG
```

Figure A.19.38 (c) Using GET\_VAR and SET\_VAR To Set Register Values

## A.19.39 %SHADOWVARS Translator Directive

**Purpose:** This directive specifies that all variables by default are created in SHADOW.

**Syntax:** %SHADOWVARS

## A.19.40 SHORT Data Type

**Purpose:** Defines a variable as a SHORT data type

**Syntax:** SHORT

**Details:**

- SHORT, is defined as 2 bytes with the range of (-32768 <= n >= 32766). A SHORT variable assigned to (32767) is considered uninitialized.
- SHORTs are allowed only within an array or within a structure.
- SHORTs can be assigned to BYTES and INTEGERS, and BYTES and INTEGERS can be assigned to SHORTs. An assigned value outside the SHORT range is detected during execution and causes the program to be aborted.

## A.19.41 SIGNAL EVENT Action

---

**Purpose:** Signals an event that might satisfy a condition handler or release a waiting program

**Syntax:** SIGNAL EVENT[event\_no]

where:

event\_no : an INTEGER expression

**Details:**

- You can use the SIGNAL EVENT action to indicate a user-defined event has occurred.
- event\_no occurs when signaled and is not remembered. Thus, if a WHEN clause has the event as its only condition, the associated actions will occur.
- If other conditions are specified that are not met at the time the event is signaled, the actions are not taken, even if the other conditions are met at another time.
- event\_no must be in the range of -32768 to 32767. Otherwise, the program is aborted with an error.

**See Also:** [Section A.5.7, EVENT Condition](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.42 SIGNAL EVENT Statement

---

**Purpose:** Signals an event that might satisfy a condition handler or release a waiting program

**Syntax:** SIGNAL EVENT [event\_no]

where:

event\_no : an INTEGER

**Details:**

- You can use the SIGNAL EVENT statement to indicate a user-defined event has occurred.
- event\_no occurs when signaled and is not remembered. Thus, if a WHEN clause has the event as its only condition, the associated actions will occur.
- If other conditions are specified that are not met at the time the event is signaled, the actions are not taken, even if the other conditions are met at another time.
- event\_no must be in the range of -32768 to 32767. Otherwise, the program is aborted with an error.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information, [Section A.5.7, EVENT Condition](#)

**Example:** Refer to the [Section A.4.18, DISABLE CONDITION Statement](#) example program.

## A.19.43 SIGNAL SEMAPHORE Action

---

**Purpose:** Adds one to the value of the indicated semaphore

**Syntax:** SIGNAL SEMAPHORE[semaphore\_no]

where:

semaphore\_no : an INTEGER expression

**Details:**

- The semaphore indicated by semaphore\_no is incremented by one.
- semaphore\_no must be in the range of 1 to the number of semaphores defined on the controller.

**See Also:** [Section 17.8, USING QUEUES FOR TASK COMMUNICATIONS](#) for more information and examples.

## A.19.44 SIN Built-In Function

**Purpose:** Returns a REAL value that is the sine of the specified angle argument

**Syntax:** SIN(angle)

Function Return Type: REAL

Input/Output Parameters:

[in] angle : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- angle specifies an angle in degrees.
- angle must be in the range of  $\pm 18000$  degrees. Otherwise, the program will be aborted with an error.

**Example:** Refer to [Section A.20.1, TAN Built-In Function](#) for a detailed program example.

## A.19.45 SQRT Built-In Function

**Purpose:** Returns a REAL value that is the positive square root of the specified REAL argument

**Syntax:** SQRT(x)

Function Return Type: REAL

Input/Output Parameters:

[in] x : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- x must not be negative. Otherwise, the program will be aborted with an error.

**Example:** The following example calculates the square root of the expression  $(a*a+b*b)$  and indicates that this is the hypotenuse of a triangle.

```
c = SQRT(a*a+b*b)
WRITE ('The hypotenuse of the triangle is ',c::6::2)
```

**Figure A.19.45 SQRT Built-In Function**

## A.19.46 %STACKSIZE Translator Directive

---

**Purpose:** Specifies stack size in long words.

**Syntax:** %STACKSIZE = n

**Details:**

- n is the stack size.
- The default value of n is 300 (1200 bytes).

**See Also:** [Section 5.1.6, Stack Usage](#) for information on computing stack size

## A.19.47 STD\_PTH\_NODE Data Type

---

**Purpose:** Defines a data type for use in PATHs.

**Syntax:** STD\_PTH\_NODE = STRUCTURE

node\_pos : POSITION in GROUP[1]

group\_data : GROUP\_ASSOC in GROUP[1] (no longer used)

common\_data : COMMON\_ASSOC (no longer used)

ENDSTRUCTURE

**Details:**

- If the NODEDATA clause is omitted from the PATH declaration, then STD\_PTH\_NODE will be the default.
- Each node in the PATH will be of this type.

## A.19.48 STOP Action

---

**Purpose:** Stops any motion in progress, leaving it in a resumable state

**Syntax:** STOP <GROUP[n{,n}]>

**Details:**

- Any motion in progress is decelerated to a stop. The unfinished motion as well as any pending motions are grouped together in a motion set and placed on a stack.
- More than one motion might be stacked by a single STOP action.
- If the KAREL program was waiting for the completion of the motion in progress, it will continue to wait.
- The stacked motion set can be removed from the stack and restarted with either a RESUME statement or action or by issuing RESUME from the operator interface (CRT/KB).
- If the group clause is not present, all groups for which the task has control (when the condition is defined) will be stopped.
- If the motion that is stopped, resumed, canceled, or held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are stopped, resumed, canceled, or held.
- Motion cannot be stopped for a different task.

**See Also:** [Section A.18.16, RESUME Statement](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.19.49 STOP Statement

**Purpose:** Stops any motion in progress, leaving it in a resumable state

**Syntax:** STOP <GROUP[n{,n}]>

**Details:**

- Any motion in progress is decelerated to a stop. The unfinished motion as well as any pending motions are grouped together in a motion set and placed on a stack.
- More than one motion might be stacked by a single STOP statement.
- If the KAREL program was waiting for the completion of the motion in progress, it will continue to wait.
- The stacked motion set can be removed from the stack and restarted with either a RESUME statement or action, or by issuing RESUME from the CRT/KB.
- If the group clause is not present, all groups for which the task has control will be stopped.
- If the motion that is stopped, resumed, canceled, or held is part of a SIMULTANEOUS or COORDINATED motion with other groups, the motions for all groups are stopped, resumed, canceled, or held.
- Motion cannot be stopped for a different task.

**See Also:** [Section A.18.15, RESUME Action](#) , [Section A.18.16, RESUME Statement](#) , [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** The following example stops motion if the digital input is ON.

```
IF DIN[2] THEN
    STOP
ENDIF
```

**Figure A.19.49 STOP Statement**

## A.19.50 STRING Data Type

**Purpose:** Defines a variable or routine parameter as STRING data type

**Syntax:** STRING[length]

where:

length : an INTEGER constant or literal

**Details:**

- length, the physical length of the string, indicates the maximum number of characters for which space is allocated for a STRING variable.
- length must be in the range 1 through 254 and must be specified in a STRING variable declaration.
- A length value is not used when declaring STRING routine parameters; a STRING of any length can be passed to a STRING parameter.

- Attempting to assign a STRING to a STRING variable that is longer than the physical length of the variable results in the STRING value being truncated on the right to the physical length of the STRING variable.
- Only STRING expressions can be assigned to STRING variables or passed as arguments to STRING parameters.
- STRING values cannot be returned by functions.
- Valid STRING operators are:
  - Relational operators (>, >=, =, <>, <, and <=)
  - Concatenation operator (+)
- STRING literals consist of a series of ASCII characters enclosed in single quotes (apostrophes). Examples are given in the following table.

**Table A.19.50 Example STRING Literals**

VALID	INVALID	REASON
`123456'	123456	Without quotes 123456 is an INTEGER literal

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.3, SAVING DATA TO THE DEFAULT DEVICE \(SAVE\\_VR.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#)

[Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN \(DCALP\\_EX.KL\)](#)

[Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM \(CPY\\_TP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.19.51 STR\_LEN Built-In Function

---

**Purpose:** Returns the current length of the specified STRING argument

**Syntax:** STR\_LEN(str)

Function Return Type: INTEGER

Input/Output Parameters:

[in] str : STRING

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the length of the STRING currently stored in the str argument, not the maximum length of the STRING specified in its declaration.

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#) for a detailed program example.

## A.19.52 STRUCTURE Data Type

**Purpose:** Defines a data type as a user-defined structure

**Syntax:** new\_type\_name = STRUCTURE

field\_name\_1: type\_name\_1

field\_name\_2: type\_name\_2

...

ENDSTRUCTURE

**Details:**

- A user-defined structure is a data type consisting of a list of component fields, each of which can be a standard data type or another, previously defined, user data type.
- When a program containing variables of user-defined types is loaded, the definitions of these types is checked against a previously created definition. If this does not exist, it is created.
- The following data types are not permitted as part of a data structure:
  - STRUCTURE definitions (types that are declared structures are permitted)
  - PATH types
  - FILE types
  - Vision types
  - Variable length arrays
  - The data structure itself, or any type that includes it, either directly or indirectly
- A variable may not be defined as a structure, but as a data type previously defined as a structure

**See Also:** [Section 2.4.2, User-Defined Data Structures](#)

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.19.53 SUB\_STR Built-In Function

**Purpose:** Returns a copy of part of a specified STRING argument

**Syntax:** SUB\_STR(src, strt, len)

Function Return Type: STRING

Input/Output Parameters:

[in] src : STRING

[in] strt : INTEGER

[in] len : INTEGER

%ENVIRONMENT Group : SYSTEM

**Details:**

- A substring of src is returned by the function.
- The length of substring is the number of characters, specified by len, that starts at the indexed position specified by strt.
- strt must be positive. Otherwise, the program is aborted with an error. If strt is greater than the length of src, then an empty string is returned.
- len must be positive. Otherwise, the program is aborted with an error. If len is greater than the declared length of the return STRING, then the returned STRING is truncated to fit.

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS \(LIST\\_EX.KL\)](#)

[Section B.8, USING THE FILE AND DEVICE BUILT-INS \(FILE\\_EX.KL\)](#)

## A.20 - T - KAREL LANGUAGE DESCRIPTION

---

### A.20.1 TAN Built-In Function

---

**Purpose:** Returns a REAL value that is the tangent of the specified REAL argument

**Syntax:** TAN(angle)

Function Return Type: REAL

Input/Output Parameters:

[in] angle : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- The value of angle must be in the range of  $\pm 18000$  degrees. Otherwise, the program will be aborted with an error.

**Example:** The following example uses the TAN built-in function to specify the tangent of the variable angle. The tangent should be equal to the SIN(angle) divided by COS(angle).

```

WRITE ('enter an angle:')
READ (angle,CR)

ratio = SIN(angle)/COS(angle)
IF ratio = TAN(angle) THEN
    WRITE ('ratio is correct',CR)
ENDIF

```

Figure A.20.1 TAN Built-In Function

### A.20.2 %TIMESLICE Translator Directive

---

**Purpose:** Supports round-robin type time slicing for tasks with the same priority

**Syntax:** %TIMESLICE = n

**Details:**

- n specifies task execution time in msec for one slice. The default value is 256 msec.
- The timeslice value must be greater than 0.
- This value is the maximum duration for executing the task continuously if there are other tasks with the same priority that are ready to run.
- This function is effective only when more than one KAREL task with the same priority is executing at the same time.
- The timeslice duration can be set during task execution by the SET\_TSK\_ATTR built-in routine.

## A.20.3 %TPMOTION Translator Directive

**Purpose:** Specifies that task motion is enabled when the teach pendant is on

**Syntax:** %TPMOTION

**Details:**

- This attribute can be set during task execution by the SET\_TSK\_ATTR built-in routine.

## A.20.4 TRANSLATE Built-In Procedure

**Purpose:** Translates a KAREL source file (.KL file type) into p-code (.PC file type), which can be loaded into memory and executed.

**Syntax:** TRANSLATE(file\_spec, listing\_sw, status)

Input/Output Parameters:

[in] file\_spec : STRING

[in] listing\_sw : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : TRANS

**Details:**

- file\_spec specifies the device, name and type of the file to translate. If no device is specified, the default device will be used. The type, if specified, is ignored and .KL is used instead.
- The p-code file will be created on the default device. The default device should be set to the ram disk.
- listing\_sw specifies whether a .LS file should be created. The .LS file contains a listing of the source lines and any errors which may have occurred. The .LS file will be created on the default device.
- The KAREL program will wait while the TRANSLATE built-in executes. If the KAREL program is paused, translation will continue until completed. If the KAREL program is aborted, translation will also be aborted and the .PC file will not be created.
- If the KAREL program must continue to execute during translation, use the KCL\_NO\_WAIT built-in instead.
- status explains the status of the attempted operation. If the number 0 is returned, the translation was successful. If not, an error occurred. Some of the status codes are shown below:

- 0 Translation was successful
- 268 Translator option is not installed
- 35084 File cannot be opened or created.KL file cannot be found or default device is not the RAM disk
- -1 Translation was not successful (Please see .LS file for details)

**Example:** The following example program will create, translate, load, and run another program called hello.

```

OPEN FILE util_file ('RW', 'hello.kl')
WRITE util_file ('PROGRAM hello', CR)
WRITE util_file ('%NOLOCKGROUP', CR)
WRITE util_file ('BEGIN', CR)
WRITE util_file (' WRITE ("hello world", CR)', CR)
WRITE util_file ('END hello', CR)
CLOSE FILE util_file
TRANSLATE('hello', TRUE, status)
IF status = 0 THEN
    LOAD('hello.pc', 0, status)
ENDIF
IF status = 0 THEN
    CALL_PROG('hello', prog_index)
ENDIF

```

**Figure A.20.4 TRANSLATE Built-In Procedure**

## A.20.5 TRUNC Built-In Function

**Purpose:** Converts the specified REAL argument to an INTEGER value by removing the fractional part of the REAL argument

**Syntax:** TRUNC(x)

Function Return Type: INTEGER

Input/Output Parameters:

[in] x : REAL

%ENVIRONMENT Group : SYSTEM

**Details:**

- The returned value is the value of x after any fractional part has been removed. For example, if x=2.3, the .3 is removed and a value of 2 is returned.
- x must be in the range of -2147483648 to +2147483583. Otherwise, the program is aborted with an error.
- ROUND and TRUNC can both be used to convert a REAL expression to an INTEGER expression.

**See Also:** [Section A.18.30, ROUND Built-In Function](#)

**Example:** The following example uses the TRUNC built-in to determine the actual INTEGER value of miles divided by hours to get mph.

```

WRITE ('enter miles driven, hours used: ')
READ (miles,hours,CR)

```

```
mph = TRUNC(miles/hours)
WRITE ('miles per hour=', mph::5)
```

Figure A.20.5 TRUNC Built-In Function

## A.21 - U - KAREL LANGUAGE DESCRIPTION

### A.21.1 UNHOLD Action

**Purpose:** Releases a HOLD of motion

**Syntax:** UNHOLD <GROUP [n{,n}]>

**Details:**

- Any motion that was in progress when the last HOLD was executed is resumed.
- If motions are not held, the action has no effect.
- Held motions are canceled if a RELEASE statement is executed.
- If the group clause is not present, all groups or which the task has control (when the condition is defined) will be resumed.
- Motion cannot be stopped for a different task.

**Example:** The following example shows a global condition handler that issues an UNHOLD action to resume robot motion is when TPIN[1] is pressed.

```
CONDITION[1] :
  WHEN TPIN[1]+ DO
    UNHOLD
  ENDCONDITION
```

Figure A.21.1 UNHOLD Action

### A.21.2 UNHOLD Statement

**Purpose:** Releases a HOLD of motion

**Syntax:** UNHOLD <GROUP [n{,n}]>

**Details:**

- Any motion that was in progress when the last HOLD was executed is resumed.
- If motions are not held, the statement has no effect.
- Held motions are canceled if a RELEASE statement is executed.
- If the group clause is not present, all groups or which the task has control (when the condition is defined) will be resumed.
- Motion cannot be stopped for a different task.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** The following example initiates a move to PR[1] and HOLDS the motion. If DIN[1] is ON, UNHOLD allows the program to resume motion.

```
move_to_pr1 — Call TP program to move to PR[1]
HOLD
IF DIN[1] THEN
    UNHOLD
ENDIF
```

**Figure A.21.2 UNHOLD Statement**

## A.21.3 UNINIT Built-In Function

---

**Purpose:** Returns a BOOLEAN value indicating whether or not the specified argument is uninitialized

**Syntax:** UNINIT(variable)

Function Return Type: BOOLEAN

Input/Output Parameters:

[in] variable : any KAREL variable

%ENVIRONMENT Group : SYSTEM

**Details:**

- A value of TRUE is returned if variable is uninitialized. Otherwise, FALSE is returned.
- variable can be of any data type except an unsubscripted ARRAY, PATH, or structure.

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#) for a detailed program example.

## A.21.4 %UNINITVARS Translator Directive

---

**Purpose:** This directive specifies that all variables are by default uninitialized.

**Syntax:** %UNINITVARS

## A.21.5 UNLOCK\_GROUP Built-In Procedure

---

**Purpose:** Unlocks motion control for the specified group of axes

**Syntax:** UNLOCK\_GROUP(group\_mask, status)

Input/Output Parameters:

[in] group\_mask : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : MULTI

**Details:**

- group\_mask specifies the group of axes to unlock for the running task. The group numbers must be in the range of 1 to the total number of groups defined on the controller.

**Table A.21.5 Group\_mask Settings**

GROUP	DECIMAL	BIT
Group 1	1	1
Group 2	2	2
Group 3	4	3

To specify multiple groups select the decimal values, shown in [Table A.21.5](#), which correspond to the desired groups. Then connect them together using the OR operator. For example to specify groups 1, 3, enter "1 OR 4".

- Unlocking a group indicates that the task is done with the motion group.
- When a task completes execution (or is aborted), all motion groups that are locked by the program will be unlocked automatically.
- If motion is executing or pending when the UNLOCK\_GROUP built-in is called, then status is set to **17039**, Executing motion exists.
- If motion is stopped when the UNLOCK\_GROUP Built-In is called, then status is set to **17040**, Stopped motion exists.
- If a motion statement is encountered in a program that has the %NOLOCKGROUP directive, the task will attempt to get motion control for all the required groups if it does not already have it. The task will pause if it cannot get motion control.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**See Also:** [Section A.12.4, LOCK\\_GROUP Built-In Procedure](#) , [Chapter 17, MULTI-TASKING](#) , for more information and examples.

**Example:** The following example unlocks groups 1, 2, and 3, and then locks group 3.

```

PROGRAM lock_grp_ex
%ENVIRONMENT MOTN
%ENVIRONMENT MULTI
VAR
    status: INTEGER
BEGIN
    REPEAT
        -- Unlock groups 1, 2, and 3
        UNLOCK_GROUP(1 OR 2 OR 4, status)
        IF status = 17040 THEN
            CNCL_STP_MTN      -- or RESUME
        ENDIF
        DELAY 500
    UNTIL status = 0
        -- Lock only group 3
        LOCK_GROUP(4, status)
    END lock_grp_ex

```

**Figure A.21.5 UNLOCK\_GROUP Built-In Procedure**

## A.21.6 UNPAUSE Action

---

**Purpose:** Resumes program execution long enough for a routine action to be executed

**Syntax:** UNPAUSE

**Details:**

- If a routine is called as an action, but program execution is paused, execution is resumed only for the duration of the routine and then is paused again.
- If more than one routine is called, all of the routines will be executed before execution is paused again.
- The resume and pause caused by UNPAUSE do not satisfy any RESUME and PAUSE conditions.

**See Also:** [Section A.18.15, RESUME Action](#) , [Section A.16.3, PAUSE Action](#)

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

## A.21.7 UNPOS Built-In Procedure

---

**Purpose:** Sets the specified REAL variables to the location (x,y,z) and orientation (w,p,r) components of the specified XYZWPR variable and sets the specified CONFIG variable to the configuration component of the XYZWPR

**Syntax:** UNPOS(posn, x, y, z, w, p, r, c)

Input/Output Parameters:

[in] posn : XYZWPR

[out] x, y, z : REAL

[out] w, p, r : REAL

[out] c : CONFIG

%ENVIRONMENT Group : SYSTEM

**Details:**

- x, y, z, w, p, and r arguments are set to the x, y, and z location coordinates and yaw, pitch, and roll orientation angles of posn.
- c returns the configuration of posn.

**Example:** The following example uses the UNPOS built-in to add 100 to the x location argument.

```
UNPOS (CURPOS,x,y,z,w,p,r,config)
next_pos = POS (x+100,y,z,w,p,r,config)
SET_POS_REG(1, next_pos, status) — Put next_pos in PR[1]
move_to_pr1 — Call TP program to move to PR[1]
```

**Figure A.21.7 UNPOS Built-In Procedure**

## A.21.8 USING ... ENDUSING Statement

---

**Purpose:** Defines a range of executable statements in which fields of a variable of a STRUCTURE type can be accessed without repeating the name of the variable.

**Syntax:** USING struct\_var{,struct\_var} DO

{statement} ENDUSING

where:

struct\_var : a variable of STRUCTURE type

statement : an executable KAREL statement

**Details:**

- In the executable statement, if the same name is both a field name and a variable name, the field name is used.
- If the same field name appears in more than one variable, the right-most variable in the USING statement is used.
- When the translator sees any field, it searches the structure type variables listed in the USING statement from right to left.

**Example:** Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.22 - V - KAREL LANGUAGE DESCRIPTION

---

### A.22.1 V\_ACQ\_VAMAP iRVision Built-In Procedure

---

**Purpose:** Acquires a 3D area map using the specified 3D area sensor tool.

**Syntax:** V\_ACQ\_VAMAP(sensor\_name, partial, center\_pos, status)

Input/Output Parameters:

[in] sensor\_name : STRING

[in] partial : BOOLEAN

[in] center\_pos : VECTOR

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- sensor\_name is the name of the 3D area sensor tool.
- partial is an internal parameter. Always set to FALSE.
- center\_pos is an internal parameter. Declare a VECTOR variable and set x, y, and z to 0.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The acquired 3D map is kept until a new 3D map for the same 3D area sensor is acquired, or V\_CLR\_VAMAP is explicitly called or the robot controller is turned off.
- This built-in is available on R-30iB and R-30iB Plus.

- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).

**Example:**

```

-----  

PROGRAM acqvamap  

-----  

%NOLOCKGROUP  

%ENVIRONMENT cvis  

%ALPHABETIZE  

%COMMENT='acquire vamap'  

%NOPAUSE = ERROR + COMMAND + TPENABLE  

VAR  

    STATUS      : INTEGER  

    sens_name   : STRING[40]  

    int_value   : INTEGER  

    real_value  : REAL  

    center      : VECTOR  

-----  

BEGIN  

    -- the name of the area sensor tool passed from a TP program  

    -- or MACRO  

    -- TP example CALL ACQVAMAP('AREASEN1')  

    GET_TPE_PRM(1,3,int_value,real_value,sens_name,STATUS)  

    -- set center pos x, y, z = 0  

    center.x = 0  

    center.y = 0  

    center.z = 0  

    -- V_ACQ_VAMAP: get 3D area map  

    V_ACQ_VAMAP(sens_name, FALSE, center, STATUS)  

    -- success if status is zero  

    IF STATUS <> 0 THEN  

        WRITE ('V_ACQ_VAMAP FAILED with ERROR CODE ', STATUS, CR)  

        ABORT  

    ENDIF  

END acqvamap
-----
```

## A.22.2 V\_ADJ\_2D *iRVision* Built-In Procedure

**Purpose:** Modifies the offset in a specified vision register by X and Y.

**Syntax:** V\_ADJ\_2D(register\_no, x, y, status)

Input/Output Parameters:

[in] register\_no : INTEGER

[in] x : REAL

[in] y : REAL

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- register\_no is the number of the vision register where the vision offset is stored.
- x is the amount in X to adjust the offset.
- y is the amount in Y to adjust the offset.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

## A.22.3 V\_CAM\_CALIB *iRVision* Built-In Procedure

**Purpose:** Finds the calibration grid for either a single plane or multiple plane calibration.

**Syntax:** V\_CAM\_CALIB (cal\_name, func\_code, status)

Input/Output Parameters:

[in] cal\_name : STRING

[in] func\_code : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group: CVIS

**Details:**

- cal\_name is the name of the calibration setup file created in setup mode.
- func\_code is the plane number to find. One (1) for first plane or single plane calibration and two (2) for second plane in multiplane calibration. See [Table A.22.3](#) .

**Table A.22.3 Function Code Values**

Calibration Type	Function Code
Grid Pattern Calibration	Specify the index of the calibration plane: 1 or 2.
Robot-generated Grid Calibration	Specify a different number for each calibration point. The robot-generated grid calibration function automatically generates a TP program that calls the instruction identical to this built-in. So this built-in will not be used for this calibration tool.
3DL Calibration	Specify the index of the calibration plane: 1 or 2.
Visual Tracking Calibration	Not supported
Simple 2D Calibration	Not supported

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The calibration file must already exist and be set up. This built-in will recalibrate an already calibrated file.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).

- For R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

**Example:**

```
-----
PROGRAM cal
-----
%NOLOCKGROUP
%ENVIRONMENT CVIS
%ALPHABETIZE
%COMMENT = 'iRVision CAL Built-in'
%NOPAUSE = ERROR + COMMAND + TPENABLE
VAR
    STATUS      : INTEGER
-----

BEGIN
-- Find calibration plane one
V_CAM_CALIB('CALL',1,STATUS)
-- success if status is zero
IF STATUS <> 0 THEN
    WRITE ('V_CAM_CALIB failed - ERROR ', STATUS, CR)
    ABORT
ENDIF

END cal
```

## A.22.4 V\_CAM\_CHECK iRVision Built-In Procedure

**Purpose:** Gets the power status of the specified camera.

**Syntax:** V\_CAM\_CHECK(camdata\_name, power\_on, status)

Input/Output Parameters:

[in] camdata\_name : STRING

[out] power\_on : BOOLEAN

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- camdata\_name is the name of the camera data.
- power\_on is camera power status. The following values are stored:
  - 1 : Camera power is ON.
  - 0 : Camera power is OFF.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

## A.22.5 V\_CLR\_VAMAP iRVision Built-In Procedure

**Purpose:** Clears the 3D area map of a specified area sensor tool from memory.

**Syntax:** V\_CLR\_VAMAP(sensor\_name, status)

Input/Output Parameters:

[in] sensor\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- sensor\_name is the name of the 3D area sensor tool.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is available on R-30iB and R-30iB Plus.
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- For R-30iB, V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).

## A.22.6 V\_CSAPI\_GETVALUE iRVision Built-In Procedure

**Purpose:** An API for creating a custom screen for iRVision. Gets the value of the specified parameter.

**Syntax:** V\_CSAPI\_GETVALUE(vp\_name, vo\_name, value, status)

Input/Output Parameters:

[in] vp\_name : STRING

[in] vo\_name : STRING

[out] value : REAL

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process that holds the parameter whose value you want to get.
- vo\_name is the name of the vision override for specifying the parameter whose value you want to get.
- value is the current value of the parameter specified by the vision override.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- If the V\_CSAPI\_SETVALUE built-in procedure has overwritten the value, the overwritten value is retrieved.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.7 V\_CSAPI\_NUMSET iRVision Built-In Procedure

---

**Purpose:** An API for creating a custom screen for iRVision. Gets the number of values that have been overwritten by the V\_CSAPI\_SETVALUE built-in procedure.

**Syntax:** V\_CSAPI\_NUMSET(vp\_name, num\_values, status)

Input/Output Parameters:

[in] vp\_name : STRING  
 [in] num\_values : REAL  
 [out] status : INTEGER  
 %ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process for which you want to obtain the information.
- num\_values stores the number of overwritten values in the specified vision process.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.8 V\_CSAPI\_RESETDATA iRVision Built-In Procedure

---

**Purpose:** An API for creating a custom screen for iRVision. Restores all the values overwritten by the V\_CSAPI\_SETVALUE built-in procedure for the specified vision process.

**Syntax:** V\_CSAPI\_RESETDATA(vp\_name, status)

Input/Output Parameters:

[in] vp\_name : STRING  
 [out] status : INTEGER  
 %ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process that holds the parameter whose value you want to restore.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.9 V\_CSAPI\_SAVEDATA iRVision Built-In Procedure

**Purpose:** An API for creating a custom screen for iRVision. Saves the values overwritten by the V\_CSAPI\_SETVALUE built-in procedure in the specified vision process.

**Syntax:** V\_CSAPI\_SAVEDATA(vp\_name, status)

Input/Output Parameters:

[in] vp\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process that you want to save.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The values overwritten by the V\_CSAPI\_SETVALUE built-in procedure are saved in the vision process.
- After saving the vision process, clear the overwritten values related to the specified vision process.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

 **CAUTION**

This built-in procedure is prepared for use from the custom screen. Please do not call many times from programs used during production. Vision data are stored in the robot controller FROM module by default. There is a limit to the number of times the data on the flash memory of the FROM can be changed, and by executing this built-in procedure frequently, the limit may be reached unexpectedly and may interfere with normal robot operation.

## A.22.10 V\_CSAPI\_SETVALUE iRVision Built-In Procedure

**Purpose:** An API for creating a custom screen for iRVision. Overwrites the value of the parameter specified by the vision override

**Syntax:** V\_CSAPI\_SETVALUE(vp\_name, vo\_name, value, status)

Input/Output Parameters:

[in] vp\_name : STRING

[in] vo\_name : STRING

[in] value : REAL

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process that holds the parameter whose value you want to overwrite.
- vo\_name is the name of the vision override for specifying the parameter whose value you want to overwrite
- value is value to overwrite the parameter specified by the vision override
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The overwritten value is temporary and does not overwrite the teaching contents of the vision process.
- The overridden value is used by the V\_CSAPI\_TESTRUN built-in procedure executed after the V\_CSAPI\_SETVALUE built-in procedure. However, executing the V\_CSAPI\_TESTRUN built-in procedure does not clear the overwritten value.
- If you want to save the overwritten value in the vision process, please execute the V\_CSAPI\_SAVEDATA built-in procedure.
- If you want to clear the overwritten values, please execute the V\_CSAPI\_RESETDATA built-in procedure.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.11 V\_CSAPI\_TESTRUN iRVision Built-In Procedure

**Purpose:** An API for creating a custom screen for iRVision. Test-runs the specified vision process using the values overwritten by the V\_CSAPI\_SETVALUE built-in procedure.

**Syntax:** V\_CSAPI\_TESTRUN(vp\_name, camera\_view, status)

Input/Output Parameters:

[in] vp\_name : STRING  
 [in] camera\_view : INTEGER  
 [out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process that you want to test run.
- camera\_view is the number of the camera view that you want to execute. Number starts from 1. A value of 0 will run all of the views.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in uses the values overwritten by the V\_CSAPI\_SETVALUE built-in procedure to perform snap and find with the vision process.
- This built-in is available on R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.12 V\_DISPLAY4D iRVision Built-In Procedure

---

**Purpose:** Displays the 4D Graphics screen for the specified vision tool on the iPendant.

**Syntax:** V\_DISPLAY4D(vistool\_name, status)

Input/Output Parameters:

[in] vistool\_name : STRING

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vistool\_name is the name of the vision tool for which the 4D Graphics should be displayed.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The vision tool specified by vistool\_name can be a camera calibration, vision process, or 3D area sensor.
- This built-in is available on R-30iB and R-30iB Plus.
- For R-30iB, this built-in is available for V8.20 and later. This built-in requires an iRVision Package Option (R685, R686, R687, R688).

## A.22.13 V\_FIND\_VIEW iRVision Built-In Procedure

---

**Purpose:** Runs vision find processing on the specified vision process, using a previously captured image. When the vision process has more than one camera view, processing is performed for the specified view.

**Syntax:** V\_FIND\_VIEW(vp\_name, camera\_view, image\_reg, status)

Input/Output Parameters:

[in] vp\_name : STRING

[in] camera\_view : INTEGER

[in] image\_reg : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process to run.
- camera\_view is the number of the camera view, if the vision process is multi-view. A value of -1 will run all of the views.
- image\_reg is the number of the image register which contains the existing image.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The robot controller must be configured to have one or more image registers. To do so, set \$VISION\_CFG.\$NUM\_IMREGS to a nonzero value. You may also need to modify the value of \$VISION\_CFG.\$IMREG\_SIZE depending on the type and resolution of the camera:

- Grayscale cameras:
  - 320 x 240 : Set *\$VISION\_CFG.\$IMREG\_SIZE* minimum value = 75
  - 640 x 480 : 300 (default)
  - 640 x 960 : 600
  - 1024 x 768 : 768
  - 1280 x 480 : 600
  - 1280 x 1024 : 1280
- Color digital cameras:
  - 640 x 480 : 1200
  - 1024 x 768 : 2304
- 3D laser sensor: 1500
- After modifying *\$VISION\_CFG.\$NUM\_IMREGS* and/or *\$VISION\_CFG.\$IMREG\_SIZE*, turn off the robot controller and then turn it back on for the changes to take effect.
- Note that increasing these values will decrease the amount of memory available for other applications.
- For R-30iA, this built-in requires the iRVision KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

## A.22.14 V\_FIND\_VLINE iRVision Built-In Procedure

---

**Purpose:** Executes and Image To Points Vision Process. This is the dedicated built-in for the Image to Points Vision Process. This outputs the view lines that go through the points extracted from the found outline of a workpiece to a PATH variable.

**Syntax:** `V_FIND_VLINE(vp_name, view_num, imreg_num, vlines, status)`

Input/Output Parameters:

[in] `vp_name` : STRING  
 [in] `view_num` : INTEGER  
 [in] `imreg_num` : INTEGER  
 [out] `vlines` : PATH  
 [out] `status` : INTEGER

%ENVIRONMENT Group : CVIS

### Details:

- `vp_name` is the name of a vision process.
- `view_num` is the number of the camera view. If the vision process is an Image To Points Vision Process, specify 1.
- `imreg_num` is the index number of the image register which stores the image. When you do not use the image register, that is, when you want to snap and find, specify 0 or a negative integer.
- `vlines` is the PATH variable that stores view lines. When this built-in executes an Image To Points Vision Process, the vision process outputs view lines which go through the points extracted from found chains to the specified PATH variable.

- The type of PATH variable for the view lines is defined in advance in VITPTYPS.KL as follows. You cannot use another type for the PATH variable. When you create KAREL programs, make them include VITPTYPS.KL and define the PATH variable for the view lines by VLINES\_T.

```
%ENVIRONMENT vitpdef

TYPE
    VLINES_T = PATH NODEDATA = VITP_ND_VL_T, PATHHEADER = VITP_HD_VL_T
```

**Figure A.22.14 (a) The type of the PATH variable for the view lines (VITPTYPS.KL)**

- The header and node of the PATH variable for the view lines is defined as follows. The view lines are represented based on the Application User Frame specified in the camera calibration.

```
TYPE
    VITP_HD_VL_T = STRUCTURE -- * Header of view lines *
        grp_num : INTEGER RW -- * Motion group number for view lines *
        ufrm_num : INTEGER RW -- * User frame for view lines *
        focal_point : VECTOR RW -- * Focal point *
    ENDSTRUCTURE -- VITP_HD_VL_T

    VITP_ND_VL_T = STRUCTURE -- * Node of view lines *
        dir : VECTOR RW -- * Direction vector of view line *
        pol : VECTOR RW -- * Edge polarity vector *
        model_id : INTEGER RW -- * ID of consecutive edge points *
    ENDSTRUCTURE -- VITP_ND_VL_T
```

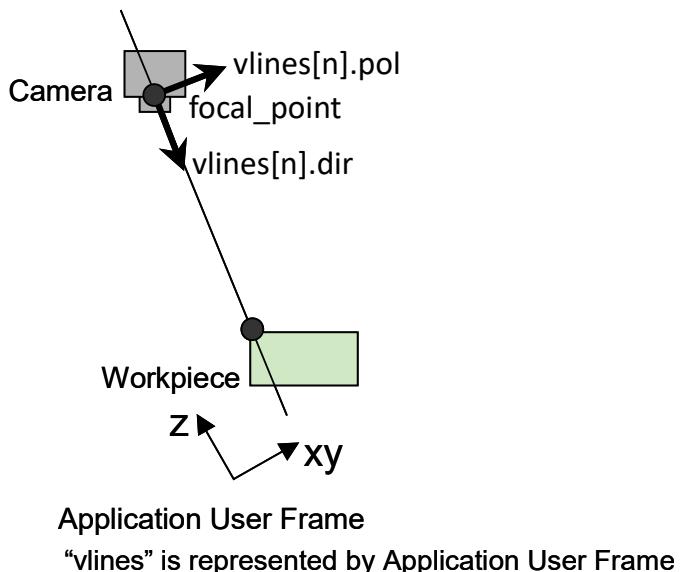
**Figure A.22.14 (b) The header and node of the PATH variable for the view lines**

- Members of VITP\_HD\_VL\_T (Header)**

- grp\_num** - The motion group's number. The view lines are calculated for the motion group.
- ufrm\_num** - The Application User Frame's number. The view lines are represented based on this frame.
- focal\_point** - The focal point of a camera (unit: mm). This is represented based on the Application User Frame.

- Members of VITP\_ND\_VL\_T (Node)**

- dir** - The direction vector of a view line. This is represented based on the Application User Frame.
- pol** - The direction vector which points from the view line's pixel toward the dark side around the pixel in an image. This vector is normal to the view line. This vector is represented based on the Application User Frame.
- model\_id** - Model ID that are assigned to the chain found by an Edge Points Locator Tool or a Selected Edge Points Locator Tool. The locator tools assign the Model ID to the found chain. Each point extracted from one chain has the same Model ID.

**Figure A.22.14 (c) Physical relationship**

- status is the process status. If the process is successful, then 0 is set as status. If the process is not successful, then an alarm code that is not equal to 0 is set.
- When the image register is not used, the next line of this built-in is executed soon after an image is snapped in the vision process. The finding and extracting process is continued in the background. So, for example, while the vision process finds chains and extracts points in the previous snapped image, you can execute KAREL programs or a robot motion instruction to the next position.
- When `INTSCT_VL_PL` which calculates the intersection points of view lines and a plane is executed during finding chains and extracting points in the vision process, the `INTSCT_VL_PL` built-in waits for the completion of the processes of the vision process.
- Each time this built-in is executed, all of the nodes of the PATH variable for the view lines are deleted and new nodes are added to the PATH variable.
- This built-in requires the iRVision KAREL Interface Software Option (J870).
- This built-in is also available on R-30iB and R-30iB Plus.
- You need the Virtual Robot with V7.70P/15 or later in order to translate this built-in.

**Example:**

- The following is the KAREL program to get the view lines.

**Figure A.22.14 (d) KAREL program to get the view lines**

```

PROGRAM vfndvl
-----
--%
%COMMENT='sample'
%NOLOCKGROUP
%RWACCESS
%ENVIRONMENT cvis
%INCLUDE vitptyps
-----
--%
VAR
-----
vpname IN CMOS : STRING[20]
viewnum IN CMOS : INTEGER

```

```
imregnum IN CMOS : INTEGER
view_lines : VLINES_T
stat : INTEGER

-----
---  

BEGIN
-----
---  

    V_FIND_VLINE(vpname, viewnum, imregnum, view_lines, stat)
    IF stat <> 0 THEN
        POST_ERR(stat, '', 0, 0)
    ENDIF
END vfndvl
```

### CAUTION

Please specify DRAM or CMOS for the PATH variable of the view lines. You cannot specify SHADOW for the PATH variable of the view lines. When you use DRAM, memory contents do not retain their stored values when it is turned off. But the processing speed is improved. On the other hand, when you use CMOS, memory contents retain their stored values when it is turned off. But the processing speed is slower.

When there are many view lines, if you use CMOS, the processing speed may be very slow. In that case, you should use DRAM. Even if you use DRAM, you can save the DRAM variable contents to Flash ROM by using SAVE\_DRAM built-in.

## A.22.15 V\_GET\_FOUND iRVision Built-In Procedure

**Purpose:** Gets the information of a found workpiece from a vision process and stores it in specified variables.

**Syntax:** V\_GET\_FOUND(vp\_name, frm\_num, model\_id, enc\_count, offset, found\_pos, meas\_val, status)

Input/Output Parameters:

[in] vp\_name : STRING  
[out] frm\_num : INTEGER  
[out] model\_id : INTEGER  
[out] enc\_count : INTEGER  
[out] offset : XYZWPR  
[out] found\_pos : ARRAY OF XYZWPR  
[out] meas\_val : ARRAY OF REAL  
[out] status : INTEGER

%ENVIRONMENT Group : CVIS

#### Details:

- vp\_name is the name of the vision process.
- frm\_num is the offset frame number of the found workpiece.
- model\_id is the model ID of the found workpiece.

- enc\_count is the encoder value that got at the moment that the image is acquired in order to find the workpiece.
- offset is the offset data of the found workpiece.
- found\_pos is the found positions in camera view 1-4.
- meas\_val is the measurement values of the found workpiece. The measurement values are specified in the measurement output tool of a vision process.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is used after V\_RUN\_FIND built-in.
- If vision program finds more than one workpiece, call this built-in repeatedly until status of this built-in is not equal to 0. When there are no more workpieces to get, the status becomes 117151.
- This built-in is available on R-30iB and R-30iB Plus.
- This built-in requires one of the following options: iRVision 2D (J901), iRVision 3DL (J902), iRVision 3D Area Sensor (J913), iRVision 3D Vision Sensor (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.16 V\_GET\_OFFSET iRVision Built-In Procedure

**Purpose:** Gets a vision offset from a vision process and stores it in a specified vision register.

**Syntax:** V\_GET\_OFFSET(vp\_name, register\_no, status)

Input/Output Parameters:

[in] vp\_name : STRING  
 [in] register\_no : INTEGER  
 [out] status : INTEGER  
 % ENVIRONMENT Group : CVIS

### Details:

- vp\_name is the name of the vision process created in setup mode
- register\_no is the register number in which the offset and vision data is placed.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This command is used after a V\_RUN\_FIND built-in procedure. If image processing is not yet completed when V\_GET\_OFFSET is executed, it waits for the completion of the image processing. V\_GET\_OFFSET stores the vision offset for a workpiece in a vision register. When the vision process finds more than one workpiece, V\_GET\_OFFSET should be called repeatedly.
- For R-30iA, this built-in requires the iRVision KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

### Example:

```
-----  

PROGRAM vision  

-----  

%NOLOCKGROUP
```

```
%ENVIRONMENT cvis
%ALPHABETIZE
%COMMENT = 'IRVision Built-in'
%NOPAUSE = ERROR + COMMAND + TPENABLE
VAR
    STATUS      : INTEGER
    visprocess  : STRING[8]
    int_value   : INTEGER
    real_value  : REAL
-----
BEGIN
    -- the name of the vision process passed from a TP program or MACRO
    -- TP example CALL VISION('VP1')
    GET_TPE_PRM(1,3,int_value,real_value,visprocess,STATUS)

    -- V_RUN_FIND, snap an image and run the vision process
    V_RUN_FIND(visprocess, 0, STATUS)

    -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_RUN_FIND FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

    -- V_GET_OFFSET, get the first offset from the run_find command
    -- put the offset into VR[1]
    -- call V_GET_OFFSET multiple times to get offsets from multiple
    -- parts
    V_GET_OFFSET(visprocess, 1, STATUS)

    -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_GET_OFFSET FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

    -- Get all the offset values from VR[1] so they can be put into
    -- a PR
    VREG_OFFSET(1,1,status)
END vision
```

## A.22.17 V\_GET\_PASSFL iRVision Built-In Procedure

**Purpose:** Gets the status of the error proofing vision process. It then stores the result in a specified numeric register.

**Syntax:** V\_GET\_PASSFL(vp\_name, register\_no, status)

Input/Output Parameters:

- [in] vp\_name : STRING
- [in] register\_no : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the error proofing or inspection vision process created in setup mode
- register\_no is the register number the command will set with the status of the error proofing/inspection operation.
  - 0 : operation failed
  - 1 : operation was successful
  - 2 : operation is undetermined because the parent tool failed
- status explains the status of the attempted operation. If status is not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the iRVision KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

**Example:**

```
-----
PROGRAM eproof
-----
%NOLOCKGROUP
%ENVIRONMENT CVIS
%ALPHABETIZE
%COMMENT = 'iRVision EP Built-in'
%NOPAUSE = ERROR + COMMAND + TPENABLE
VAR
    STATUS      : INTEGER
-----

BEGIN
    -- Run error proofing vision process 'EP1'
    V_RUN_FIND('EP1',0,STATUS)
        -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_RUN_FIND FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

    -- Put error proofing result into R[1]
    V_GET_PASSFL('EP1',1,STATUS)
        -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_GET_PASSFL FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

END eproof
```

**A.22.18 V\_GET\_READ iRVision Built-In Procedure**

**Purpose:** Get a result string of a reader vision process and stores it in a specified variables.

**Syntax:** V\_GET\_READ(vp\_name, sr\_num, filename, str\_len, status)

Input/Output Parameters:

```
[in] vp_name : STRING
[in] sr_num : INTEGER
[in] filename : STRING
[out] str_len : INTEGER
[out] status : INTEGER
%ENVIRONMENT Group : CVIS
```

**Details:**

- vp\_name is the name of the vision process
- sr\_num is the string register number that stores the read result. If 0 is specified, the read result is stored in the file specified by filename instead of the STRING register.
- filename is the file name to write the read result. It is used only when 0 is specified for sr\_num.
- str\_len is the string length stored in the STRING register
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- This built-in is available on R-30iB and R-30iB plus.
- This built-in requires one of the following options: *iRVision 2D* (J901), *iRVision 3DL* (J902), *iRVision 3D Area Sensor* (J913), *iRVision 3D Vision Sensor* (J914).

**Example:** Please refer to [Section B.14, iRVISION CUSTOM SCREEN USING V\\_CSAPI BUILT-INS](#) for detailed program examples.

## A.22.19 V\_GET\_VPARAM *iRVision* Built-In Procedure

**Purpose:** Retrieves the value of a vision parameter from a specified vision process. When the vision process has more than one camera view, the value is retrieved for the specified view.

**Syntax:** V\_GET\_VPARAM(vp\_name, param\_no, camera\_view, reg\_no, status)

Input/Output Parameters:

```
[in] vp_name : STRING
[in] param_no : INTEGER
[in] camera_view : INTEGER
[in] reg_no : INTEGER
[out] status : INTEGER
%ENVIRONMENT Group : CVIS
```

**Details:**

- vp\_name is the name of the vision process to run.
- param\_no is the number of the parameter to retrieve. The valid values of param\_no are:
  - 0 : Number of found results
- camera\_view is the number of the camera view, if the vision process is multi-view. A value of -1 will run all of the views.
- reg\_no is the number of the register in which to store the result. The result is stored as a REAL value.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).

- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

## A.22.20 V\_IRCONNECT *iRVision* Built-In Procedure

---

**Purpose:** Sends the most recent *iRVision* result to mobile devices using *iRConnect*.

**Syntax:** V\_IRCONNECT(desc\_string, priority, vis\_reg\_no, status)

Input/Output Parameters:

[in] desc\_string : STRING

[in] priority : INTEGER

[in] vis\_reg\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

### Details:

- desc\_string is the description string to associate with the vision result data.
- priority is the priority level of the vision result data. The valid values of priority are:
  - 0 : Data is informational only. Not highlighted in the mobile app.
  - 1 : Data is informational only. Highlighted in green.
  - 2 : Data indicates a potential problem or warning. Highlighted in yellow.
  - 3 : Data indicates a serious issue. Highlighted in red.
- vis\_reg\_no is currently an unused parameter. Always set this parameter to 0.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The vision result data sent to the mobile device contains the same information as the *iRVision Runtime* screen, including
  - Vision process name
  - Status
  - Number of found results
  - Found positions
  - Time to find
  - Result score, model ID, fit error, etc.
  - Result measurements
- Runtime images may also be sent via *iRConnect* if enabled in the Vision Configuration screen.
- This built-in is available on R-30iB and R-30iB Plus.
- For R-30iB, V\_IRCONNECT is available for V8.20 and later. This built-in requires an *iRVision Package Option* (R685, R686, R687, R688).

## A.22.21 V\_LED\_OFF iRVision Built-In Procedure

**Purpose:** Turns off an attached LED light for vision. The LED light must be attached to a multiplexer for analog cameras.

**Syntax:** V\_LED\_OFF(status)

Input/Output Parameters:

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- V\_LED\_OFF turns off the LED light that was most recently turned on by the V\_LED\_ON built-in procedure or IRVLEDON KAREL program call.
- Only one LED light can be turned on at any time, even if multiple LED lights are connected to the multiplexer.
- This built-in is available on R-30iB.
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).

## A.22.22 V\_LED\_ON iRVision Built-In Procedure

**Purpose:** Turns on an LED light attached for vision. The LED light must be connected to a multiplexer for analog cameras.

**Syntax:** V\_LED\_ON(channel, intensity, status)

Input/Output Parameters:

[in] channel : INTEGER

[in] intensity : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- channel is the multiplexer channel of the attached LED light. Valid values are 1 to 8.
- intensity is the desired intensity level to set the LED light to. Valid values are 1 to 16.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- Only one LED light can be turned on at any time, even if multiple LED lights are connected to the multiplexer. Turning on a second LED light will automatically turn off the first LED light. Some delay may occur between the first light turning off and the second light turning on.
- This built-in is available on R-30iB.
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).

- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).

## A.22.23 V\_OVERRIDE *iRVision* Built-In Procedure

---

**Purpose:** Sets the value of the specified Vision Override tool.

**Syntax:** V\_OVERRIDE(*ovrd\_name*, *value*, *status*)

Input/Output Parameters:

[in] *ovrd\_name* : STRING

[in] *value* : REAL

[out] *status* : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- *ovrd\_name* is the name of the Vision Override tool to set. The Vision Override tool is created and configured in *iRVision Setup*.
- *value* is the value to change the Vision Override tool property to.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.
- *V\_OVERRIDE* is used to temporarily modify a property that was saved with a vision process.
- Generally, this command is used immediately before performing a *V\_RUN\_FIND* operation or other vision process execution.
- Once the vision process find has been executed, the values that were set by *V\_OVERRIDE* are reset to their saved values.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

## A.22.24 V\_RUN\_FIND *iRVision* Built-In Procedure

---

**Purpose:** Starts an *iRVision* process. When a specified vision process has more than one camera view, location is performed for the specified camera views.

**Syntax:** V\_RUN\_FIND(*vp\_name*, *camera\_view*, *status*)

Input/Output Parameters:

[in] *vp\_name* : STRING

[in] *camera\_view* : INTEGER

[out] *status* : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process created in setup mode
- camera\_view is the number of the camera view. Used for multi camera vision processes. A value of -1 will run all of the views.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the iRVision KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the iRVision Client Option (J917) or iRVision VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an iRVision Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

**Example:**

```
-----
PROGRAM vision
-----
%NOLOCKGROUP
%ENVIRONMENT cvis
%ALPHABETIZE
%COMMENT = 'iRVision Built-in'
%NOPAUSE = ERROR + COMMAND + TPENABLE
VAR
    STATUS      : INTEGER
    visprocess  : STRING[8]
    int_value   : INTEGER
    real_value  : REAL
-----

BEGIN
    -- the name of the vision process passed from a TP program or MACRO
    -- TP example CALL VISION('VP1')
    GET_TPE_PRM(1,3,int_value,real_value,visprocess,STATUS)

    -- V_RUN_FIND, snap an image and run the vision process
    V_RUN_FIND(visprocess, 0, STATUS)

    -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_RUN_FIND FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

    -- V_GET_OFFSET, get the first offset from the run_find command
    -- put the offset into VR[1]
    -- call V_GET_OFFSET multiple times to get offsets from multiple
    -- parts
    V_GET_OFFSET(visprocess, 1, STATUS)

    -- success if status is zero
    IF STATUS <> 0 THEN
        WRITE ('V_GET_OFFSET FAILED with ERROR CODE ', STATUS, CR)
        ABORT
    ENDIF

    -- Get all the offset values from VR[1] so they can be put
    -- into a PR
```

```
VREG_OFFSET(1,1,status)
END vision
```

## A.22.25 V\_SAVE\_IMREG iRVision Built-In Procedure

**Purpose:** Saves an image from an image register to a file.

**Before V9.40P/31:**

**Syntax:** V\_SAVE\_IMREG(image\_reg, value, status, not\_zip)

Input/Output Parameters:

- [in] image\_reg : INTEGER
- [in] output\_path : STRING
- [out] status : INTEGER

**V9.40P/31 and later:**

**Syntax:** V\_SAVE\_IMREG(ipc\_name, image\_reg, value, status, not\_zip)

Input/Output Parameters:

- [in] ipc\_name : STRING
- [in] image\_reg : INTEGER
- [in] output\_path : STRING
- [out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- ipc\_name is the name of the target FANUC iPC device. If you are not connecting to a FANUC iPC, set this parameter to "".

### NOTE

This parameter is only used for robot software versions V9.40P/31 and later.

- image\_reg is the number of the image register which contains the existing image.
- output\_path is the path (device, directory, and file name) where the image will be saved. The file name must end with .PNG or .BMP.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The robot controller must be configured to have one or more image registers. To do so, set \$VISION\_CFG.\$NUM\_IMREGS to a nonzero value. You may also need to modify the value of \$VISION\_CFG.\$IMREG\_SIZE depending on the type and resolution of the camera:
  - Grayscale cameras:
    - 320 x 240 : Set \$VISION\_CFG.\$IMREG\_SIZE minimum value = 75
    - 640 x 480 : 300 (default)
    - 640 x 960 : 600
    - 1024 x 768 : 768
    - 1280 x 480 : 600

- 1280 x 1024 : 1280
- Color digital cameras:
  - 640 x 480 : 1200
  - 1024 x 768 : 2304
  - 3D laser sensor: 1500
- After modifying `$VISION_CFG.$NUM_IMREGS` and/or `$VISION_CFG.$IMREG_SIZE`, turn off the robot controller and then turn it back on for the changes to take effect.
- Note that increasing these values will decrease the amount of memory available for other applications.
- `V_SAVE_IMREG` is typically used in conjunction with `V_SNAP_VIEW`, which captures an image and stores it in an image register.
- `V_SAVE_IMREG` supports saving PNG or BMP image files. Specifying any other image format using `output_path` will result in a nonzero status.
- `V_SAVE_IMREG` is available for R-30iB V8.30 and later. This built-in requires an *iRVision* Package Option (R685, R686, R687, R688).
- This built-in is also available on R-30iB Plus.

## A.22.26 V\_SET\_REF *iRVision* Built-In Procedure

**Purpose:** Sets the reference position in the specified vision process after `V_RUN_FIND` has been run.

**Syntax:** `V_SET_REF(vp_name, status)`

**Input/Output Parameters:**

[in] `vp_name` : STRING

[out] `status` : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- `vp_name` is the name of the vision process created in setup mode.
- `status` explains the status of the attempted operation. If not equal to 0, then an error occurred.
- If a vision process remains open on the setup PC when `SET_REFERENCE` is executed for the vision process, the reference position cannot be written to the vision process, which results in an alarm. Close the setup window, then re-execute the command.
- When the vision process finds more than one workpiece, the position of the workpiece having the highest score is set as the reference position.
- It is recommended that only one workpiece be placed within the camera view so that an incorrect position is not set as the reference position.
- For R-30iA, this built-in requires the *iRVision* KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision* Client Option (J917) or *iRVision* VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision* Package Option (R685, R686, R687, R688).
- This built-in is also on R-30iB Plus.

## A.22.27 V\_SNAP\_VIEW *iRVision* Built-In Procedure

---

**Purpose:** Acquires an image using the specified vision process and stores it in an image register. When the vision process has more than one camera view, the image is acquired using the specified view.

**Syntax:** V\_SNAP\_VIEW(vp\_name, camera\_view, image\_reg, status)

Input/Output Parameters:

- [in] vp\_name : STRING
- [in] camera\_view : INTEGER
- [in] image\_reg : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- vp\_name is the name of the vision process to use to acquire the image.
- camera\_view is the number of the camera view, if the vision process is multi-view.
- image\_reg is the number of the image register where the image is to be stored.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- The robot controller must be configured to have one or more image registers. To do so, set *\$VISION\_CFG.\$NUM\_IMREGS* to a nonzero value. You may also need to modify the value of *\$VISION\_CFG.\$IMREG\_SIZE* depending on the type and resolution of the camera:
  - Grayscale cameras:
    - 320 x 240 : Set *\$VISION\_CFG.\$IMREG\_SIZE* minimum value = 75
    - 640 x 480 : 300 (default)
    - 640 x 960 : 600
    - 1024 x 768 : 768
    - 1280 x 480 : 600
    - 1280 x 1024 : 1280
  - Color digital cameras:
    - 640 x 480 : 1200
    - 1024 x 768 : 2304
  - 3D laser sensor: 1500
- After modifying *\$VISION\_CFG.\$NUM\_IMREGS* and/or *\$VISION\_CFG.\$IMREG\_SIZE*, turn off the robot controller and then turn it back on for the changes to take effect.
- Note that increasing these values will decrease the amount of memory available for other applications.
- For R-30iA, this built-in requires the *iRVision* KAREL Interface Software Option (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision* Client Option (J917) or *iRVision* VisTrack Package Option (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision* Package Option (R685, R686, R687, R688).
- This built-in is also on R-30iB Plus.

## A.22.28 VAR\_INFO Built-In Procedure

**Purpose:** Allows a KAREL program to determine data type and numerical information regarding internal or external program variables

**Syntax:** VAR\_INFO(prog\_name, var\_name, uninit, type\_nam, type\_value, dims, slen, status)

Input/Output Parameters:

[in] prog\_name : STRING

[in] var\_name : STRING

[out] uninit\_b : BOOLEAN

[out] type\_nam : STRING

[out] dims : ARRAY[3] OF INTEGER

[out] type\_value : INTEGER

[out] status : INTEGER

[out] slen : INTEGER

%ENVIRONMENT Group : BYNAM

**Details:**

- prog\_name specifies the name of the program that contains the specified variable. If prog\_name is blank, then it defaults to the current program being executed.
- var\_name must refer to a static program variable.
- var\_name can contain node numbers, field names, and/or subscripts.
- uninit\_b will return a value of TRUE if the variable specified by var\_name is uninitialized and FALSE if the variable specified by var\_name is initialized.
- type\_nam returns a STRING specifying the type name of var\_name.
- type\_value returns an INTEGER corresponding to the data type of var\_name. The following table lists valid data types and their representative INTEGER values.

Table A.22.28 Valid Data Types

Data Type	Value
POSITION	1
XYZWPR	2
XYZWPREXT	6
INTEGER	16
REAL	17
BOOLEAN	18
VECTOR	19
VIS_PROCESS	21
MODEL	22
SHORT	23
BYTE	24
JOINTPOS1	25

Data Type	Value
CONFIG	28
FILE	29
PATH	31
CAM_SETUP	32
JOINTPOS2	41
JOINTPOS3	57
JOINTPOS4	73
JOINTPOS5	89
JOINTPOS6	105
JOINTPOS7	121
JOINTPOS8	137
JOINTPOS9	153
JOINTPOS	153
STRING	209
user-defined type	210

- dims returns the dimensions of the array, if any. The size of the dims array should be 3.
  - dims[1]=0 if not an array
  - dims[2]=0 if not a two-dimensional array
  - dims[3]=0 if not a three-dimensional array
- slen returns the declared length of the variable specified by var\_name if it is a STRING variable.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.

**Example:** The following example retrieves information regarding the variable counter, located in util\_prog, from within the program task.

```

PROGRAM util_prog
  VAR
    counter, i : INTEGER
  BEGIN
    counter = 0
    FOR i = 1 TO 10 DO
      counter = counter + 1
    ENDFOR
  END util_prog
PROGRAM task
  VAR
    uninit_b          : BOOLEAN
    type_name         : STRING[12]
    type_code         : INTEGER
    slen, status      : INTEGER
    alen              : ARRAY[3] OF INTEGER
  BEGIN
    VAR_INFO('util_prog', 'counter', uninit_b, type_name, type_code,
             alen, slen, status)
    WRITE('counter : ', CR)
  
```

```

    WRITE('UNINIT : ', uninit_b, '      TYPE : ', type_name, cr)
END task

```

**Figure A.22.28 VAR\_INFO Built-In Procedure**

## A.22.29 VAR\_LIST Built-In Procedure

**Purpose:**Locates variables in the specified KAREL program with the specified name and data type

**Syntax:** VAR\_LIST(prog\_name, var\_name, var\_type, n\_skip, format, ary\_nam, n\_vars, status)

Input/Output Parameters:

[in] prog\_name : STRING  
[in] var\_name : STRING  
[in] var\_type : INTEGER  
[in] n\_skip : INTEGER  
[in] format : INTEGER  
[out] ary\_nam : ARRAY of STRING  
[out] n\_vars : INTEGER  
[out] status : INTEGER

%ENVIRONMENT Group BYNAM

**Details:**

- prog\_name specifies the name of the program that contains the specified variables. prog\_name can be specified using the wildcard (\*) character, which specifies all loaded programs.
- var\_name is the name of the variable to be found. var\_name can be specified using the wildcard (\*) character, which specifies that all variables for prog\_name be found.
- var\_type represents the data type of the variables to be found. The following is a list of valid data types:

**Table A.22.29 Valid Data Types**

Data Type	Value
All variable types	0
POSITION	1
XYZWPR	2
INTEGER	16
REAL	17
BOOLEAN	18
VECTOR	19
VIS_PROCESS	21
MODEL	22
SHORT	23

Data Type	Value
BYTE	24
JOINTPOS1	25
CONFIG	28
FILE	29
PATH	31
CAM_SETUP	32
JOINTPOS2	41
JOINTPOS3	57
JOINTPOS4	73
JOINTPOS5	89
JOINTPOS6	105
JOINTPOS7	121
JOINTPOS8	137
JOINTPOS9	153
JOINTPOS	153
STRING	209
user-defined type	210

- n\_skip is used when more variables exist than the declared length of ary\_nam. Set n\_skip to 0 the first time you use VAR\_LIST. If ary\_nam is completely filled with variable names, copy the array to another ARRAY of STRINGS and execute the VAR\_LIST again with n\_skip equal to n\_vars. The call to VAR\_LIST will skip the variables found in the first pass and locate only the remaining variables.
- format specifies the format of the program name and variable name. The following values are valid for format :
  - 1 = prog\_name only, no blanks
  - 2 = var\_name only, no blanks
  - 3 = [prog\_name]var\_name, no blanks
  - 4 = 'prog\_name var\_name'
  - Total length = 27 characters
  - prog\_name starts with character 1.
  - var\_name starts with character 16.
- ary\_nam is an ARRAY of STRINGS to store the variable names. If the declared length of the STRING in ary\_nam is not long enough to store the formatted data, then status is returned with an error.
- n\_vars is the number of variables stored in ary\_name.
- status will return zero if successful.

See Also: [Section A.6.2, FILE\\_LIST Built-In Procedure](#) , [Section A.16.23, PROG\\_LIST Built-In Procedure](#)

Example: Refer to [Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#), for a detailed program example.

## A.22.30 VECTOR Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as VECTOR data type

**Syntax:** VECTOR

**Details:**

- A VECTOR consists of three REAL values representing a location or direction in three dimensional Cartesian coordinates.
- Only VECTOR expressions can be assigned to VECTOR variables, returned from VECTOR function routines, or passed as arguments to VECTOR parameters.
- Valid VECTOR operators are:
  - Addition (+) and subtraction (-) mathematical operators
  - Equal (=) and the not equal ( $\neq$ ) relational operators
  - Cross product (#) and the inner product (@) operators.
  - Multiplication (\*) and division (÷) operators
  - Relative position (: ) operator
- Component fields of VECTOR variables can be accessed or set as if they were defined as follows:

```
VECTOR = STRUCTURE
  X:  REAL
  Y:  REAL
  Z:  REAL
ENDSTRUCTURE
Note: All fields are read-write
```

**Figure A.22.30 () VECTOR Data Type**

**Example:** The following example shows VECTOR as variable declarations, as parameters in a routine, and as a function routine return type.

```
VAR
  direction, offset : VECTOR
ROUTINE calc_offset(offset_vec:VECTOR):VECTOR FROM util_prog
```

**Figure A.22.30 (b) VECTOR Data Type**

## A.22.31 VOL\_SPACE Built-In Procedure

**Purpose:** Returns the total bytes, free bytes, and volume name for the specified device

**Syntax:** VOL\_SPACE(device, total, free, volume)

Input/Output Parameters:

[in] device : STRING  
[out] total : INTEGER  
[out] free : INTEGER  
[out] volume : STRING  
%ENVIRONMENT Group : FLBT

**Details:**

- device can be:
  - RD: The RAM disk returns all three parameters, but the volume name is " since it is not supported. The RAM disk must be mounted in order to query it.
  - FR: The FROM disk returns all three parameters, but the volume name is " since it is not supported. The FROM disk must be mounted in order to query it.
  - FR: Size of the Flash ROM. This only sets the total parameter.
  - DRAM: Size of the DRAM. This only sets the total parameter.
  - CMOS: Size of the CMOS ROM. This only sets the total parameter.
  - TPP: The area of system memory where teach pendant programs are stored.
  - PERM: The area of permanent CMOS RAM memory where system variables and selected KAREL variables are stored.
  - TEMP: The area of temporary DRAM memory used for loaded KAREL programs, KAREL variables, program execution information, and system operations.
  - SYSTEM: The area of temporary DRAM memory where the system software and options are stored. This memory is saved to FROM and restored on power up.

**NOTE**

All device names must end with a colon (:).

- total is the original size of the memory, in bytes.
- free is the amount of available memory, in bytes.
- volume is the name of the storage device used.

**See Also:** [Section 1.3.1, Memory](#), and your application-specific *Setup and Operations Manual* or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).

**Example:** The following example gets information about the different devices.

```

PROGRAM space
%NOLOCKGROUP
%ENVIRONMENT FLBT
VAR
  total:  INTEGER
  free:   INTEGER
  volume: STRING [30]
BEGIN
  VOL_SPACE('rd:', total, free, volume)
  VOL_SPACE('fr:', total, free, volume)
  VOL_SPACE('frp:', total, free, volume)
  VOL_SPACE('fr:', total, free, volume)
  VOL_SPACE('dram:', total, free, volume)
  VOL_SPACE('cmos:', total, free, volume)
  VOL_SPACE('tpp:', total, free, volume)
  VOL_SPACE('perm:', total, free, volume)
  VOL_SPACE('temp:', total, free, volume)
END space

```

**Figure A.22.31 VOL\_SPACE Built-In Procedure**

## A.22.32 VREG\_FND\_POS iRVision Built-in Procedure

**Purpose:** Populates the specified position register with the found position data in the specified vision register.

Syntax: VREG\_FND\_POS (visreg\_no, camera\_view, posreg\_no, status)

Input/Output Parameters:

[in] visreg\_no : INTEGER

[in] camera\_view : INTEGER

[in] posreg\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- visreg\_no is the vision register (VR) number that contains the offset data.
- camera\_view is the specified camera view for a multi view vision process.
- posreg\_no is the position register (PR) number that is to be populated with the offset data.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).
- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).
- This built-in is also on R-30iB Plus.

## A.22.33 VREG\_OFFSET iRVision Built-in Procedure

**Purpose:** Populates the specified position register with the offset data in the specified vision register.

Syntax: VREG\_OFFSET(visreg\_no, posreg\_no, status)

Input/Output Parameters:

[in] visreg\_no : INTEGER

[in] posreg\_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- visreg\_no is the vision register (VR) number that contains the offset data
- posreg\_no is the position register (PR) number that is to be populated with the offset data.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred.
- For R-30iA, this built-in requires the *iRVision KAREL Interface Software Option* (J870).
- For R-30iB before V8.20, this built-in requires the *iRVision Client Option* (J917) or *iRVision VisTrack Package Option* (R687).

- For R-30iB V8.20 and later, this built-in requires an *iRVision Package Option* (R685, R686, R687, R688).
- This built-in is also on R-30iB Plus.

## A.22.34 VT\_ACK\_QUEUE *iRVision* Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It acknowledges how a workpiece allocated by *VT\_GET\_QUEUE* call was handled to a specified work area.

**Syntax:** *VT\_ACK\_QUEUE(area\_num, vreg\_num, ack, status)*

Input/Output Parameters:

[in] *area\_num* : INTEGER

[in] *vreg\_num* : INTEGER

[in] *ack* : INTEGER

[out] *status* : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as *area\_num*. You can get the work area number from the work area name using *VT\_GET\_AREID*.
- Specify the vision register number that stores information on an allocated workpiece as *vreg\_num*.
- Specify one of the following values as *ack*:
  - 1 – Specified when the allocated workpiece has been handled correctly. This applies when, for example, the robot picks up a workpiece from a conveyor or places a workpiece on a conveyor.
  - 2 – Specified when the allocated workpiece has not been handled according to plan. This applies when, for example, the position or model ID of the allocated workpiece is evaluated by the program of the robot and picking up the workpiece is canceled intentionally. In this case, the information of the allocated workpiece is returned to the queue so that this workpiece can be handled by a downstream robot.
  - 3 – Specified when the handling of the allocated workpiece has been failed. This applies when, for example, the allocated workpiece is dropped down, and then vacuum confirmation of the gripper shows an unacceptable result. In this case, because the robot has moved the workpiece, a downstream robot cannot handle this workpiece. So, the information of the workpiece is not returned to the queue.
- The status of the process is returned to *status*. If the process is successful, then *status* is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the *iRVision KAREL interface option* (J870) and the *iRVision Tracking Queue option* (J874), or *PickTool + iRVision* (R662), or *PickTool – iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.35 VT\_CLR\_QUEUE *iRVision* Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It clears information of workpieces present in a specified work area. When called, all the information of the workpieces in the work area is entirely erased. Usually, this built-in is called just once when the system is started.

**Syntax:** VT\_CLR\_QUEUE(area\_num, status)

Input/Output Parameters:

[in] area\_num : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num. You can get the work area number from the work area name using VT\_GET\_AREID.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.36 VT\_DELETE\_PQ iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking customization. This function deletes a part from a queue. The part is identified by a work ID.

 **CAUTION**

This built-in cannot be used together with Load Balance function. This built-in cannot delete a cell of a tray by default. If you want to delete a cell of a tray, you set a flag to the system variable \$VTLINE[n].\$FLAG in all robots before you start the robot controller program. You will be able to delete a cell of a tray in V7.70P/30 and later.

1. Find the system variable \$VTLINE[n] whose \$NAME is identical to the name of the line where the work area belongs.
2. Divide \$VTLINE[n].\$FLAG by 128 and get the quotient of the division.
3. If the quotient is an even number, add 128 to \$VTLINE[n].\$FLAG to enable the function. If the quotient is an odd number, do nothing because the function has already been enabled.
4. This setting is required for all robots.

**Syntax:** VT\_DELETE\_PQ (area\_num, work\_id, status)

Input/Output Parameters:

[in] area\_num : INTEGER

[in] work\_id : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num in order to identify the queue of the work area. You can get the work area number from the work area name using VT\_GET\_AREID.

- Specify the index number of a part of the queue as work\_id. You can know the index number by reading the PATH variable's nodes gotten by VT\_READ\_PQ. Sensor task assigns unique index numbers for each found part. The specified part is deleted.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.
- You need the Virtual Robot with V7.70P/23 or later in order to translate this built-in.

**Example:** Delete a part in a queue.

```
-----
PROGRAM deletepq
-----
%COMMENT='sample'
%NOLOCKGROUP
%ENVIRONMENT cvis
%INCLUDE vstktyps
%INCLUDE vierrdef
-----
CONST
-----
-- Error Code
ER_SUCCESS = 0
-----
VAR
-----
    vtpartq FROM makepq : VTPARTQ_T
    area_id      : INTEGER
    work_id      : INTEGER
    stat         : INTEGER
    severity     : INTEGER
    err_code     : INTEGER
    i            : INTEGER
-----
BEGIN
-----
    FOR i = 1 TO vtpartq.num_parts DO
        -- Select the specific part
        if vtpartq[i].model_id = 1 THEN
            -- Delete the specific part
            area_id = vtpartq.area_id
            work_id = vtpartq[i].work_id
            VT_DELETE_PQ(area_id, work_id, stat)
            IF (stat <> ER_SUCCESS) THEN
                severity = TRUNC(stat / 10000000)
                err_code = stat - severity * 10000000
                POST_ERR(err_code, '', 0, 1)
            ABORT
        ENDIF
    ENDIF
    ENDFOR
END deletepq
```

## A.22.37 VT\_GET\_AREID iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It returns the work area number got from a specified work area name.

**Syntax:** VT\_GET\_AREID(area\_name)

Function Return Type: INTEGER

Input/Output Parameters:

[in] area\_name : STRING

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area name as area\_name.
- This built-in returns a work area number that is specified as an argument for the visual tracking built-ins such as VT\_ACK\_QUEUE.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.38 VT\_GET\_FOUND iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It outputs the information of a workpiece that a vision program finds.

**Syntax:** VT\_GET\_FOUND(vp\_name, model\_id, enc\_count, offset, found\_pos, meas\_val, status)

Input/Output Parameters:

[in] vp\_name : STRING

[out] model\_id : INTEGER

[out] enc\_count : INTEGER

[out] offset : XYZWPR

[out] found\_pos : ARRAY[4] OF XYZWPR

[out] meas\_val : ARRAY[10] OF REAL

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a vision program name as vp\_name.
- The model ID of the found workpiece is returned to model\_id.
- The encoder value got at the moment that the image is acquired in order to find the workpiece is returned to enc\_count.
- The offset data of the found workpiece is returned to offset.
- The found positions of camera view 1-4 are returned to found\_pos[1-4].

- The measurement values of the found workpiece are returned to meas\_val[1-10]. The measurement values are specified in the measurement output tool of a vision program.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in is used after V\_RUN\_FIND built-in.
- If vision program finds more than one workpiece, call this built-in repeatedly until status of this built-in is not equal to 0. When there is no more workpiece to get, the status becomes 117151.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.39 VT\_GET\_LINID iRVision Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It returns the line number got from a specified line name.

**Syntax:** VT\_GET\_LINID(line\_name)

Function Return Type: INTEGER

Input/Output Parameters:

[in] line\_name : STRING

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a line name as line\_name.
- This built-in returns the line number that is specified as an argument for VT\_PUT\_QUEUE.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.40 VT\_GET\_PFRT iRVision Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It returns the rate at which the workpieces flow through the work area in one minute (expected workpiece flow rate).

**Syntax:** VT\_GET\_PFRT(area\_num, consecutive, num\_consc, model\_id, pfrt, status)

Input/Output Parameters:

[in] area\_num : INTEGER

[in] consecutive : INTEGER

[in] num\_consc : INTEGER

[in] model\_id : INTEGER

[out] pfrt : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num. You can get the work area number from the work area name using VT\_GET\_AREID.
- Usually, specify 1 as consecutive. If workpieces are to be allocated successively regardless of the load balance to each work area, which is specified in the Line setup page, and you need the expected workpiece flow rate based on such allocation, specify the order of the next workpiece among workpieces allocated successively at once.
- Usually, specify 1 as num\_conseq. If workpieces are to be allocated successively regardless of the load balance to each work area, which is specified in the Line setup page, and you need the expected workpiece flow rate based on such allocation, specify the total number of workpieces to be allocated successively at once.
- When the expected workpiece flow rate is to be gotten by specifying a particular model ID, specify the model ID as model\_id. Usually, this argument is not specified. If no model ID is specified, set an uninitialized variable to this argument.
- The expected workpiece flow rate is stored in pfrt and calculated as follows.

Expected workpiece flow rate (workpieces/min) = Expected number of workpieces (workpieces) \* Conveyor speed (mm/sec) \* 60 (sec/min)  
/ Distance that corresponds to the work area (mm)

Where Expected number of workpieces and Distance that corresponds to the work area are calculated as follows.

- For each work area, the region for calculation is defined. The expected number of workpieces is calculated by use of workpieces contained only in this region. The downstream boundary of this region is the discard line of the work area. The upstream boundary of this region is any one of the following three:
  - 1 – If the work area is the most upstream one, the upstream boundary is the most upstream detection position of the workpieces which have been held by the work area when this built-in is called for the first time with the work area holding at least one workpiece.
  - 2 – If the work area is not the most upstream one and the Y Sort is disabled, the upstream boundary is the discard line of the previous work area in the line.
  - 3 – If the work area is not the most upstream one and the Y Sort is enabled, the upstream boundary is the position more downstream by the X Tolerance of the Y Sort from the discard line of the previous work area in the line.
- Expected number of workpieces: This is the number of workpieces which are contained in the region for calculation and expected to be handled at the work area on the basis of the settings of Load Balance of the line, Y Sort of the work area and Seq of the tray pattern and the current performance. For example, if the setting of Load Balance is 50% at the most upstream work area, and before now 4 workpieces out of 7 have been handled at this work area, and 3 workpieces are contained in the region for calculation, then the expected number of workpieces is 1 ( $= (7+3)/2 - 4$ ).
- Distance that corresponds to the work area: This is the distance between the upstream and downstream boundaries of the region for calculation of the expected number of workpieces.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.41 VT\_GET\_QUEUE iRVision Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It gets the information of one workpiece from a specified work area. The information of the gotten workpiece is stored in a vision register. The value of the encoder for the gotten workpiece is set as the trigger of a tracking motion. When there is no workpiece to be handled in the work area, the robot waits by a specified time until a workpiece actually reaches the work area.

**Syntax:** VT\_GET\_QUEUE(area\_num, vreg\_num, timeout, consecutive, model\_id, work\_id, status)

Input/Output Parameters:

- [in] area\_num : INTEGER
- [in] vreg\_num : INTEGER
- [in] timeout : INTEGER
- [in] consecutive : INTEGER
- [in] model\_id : INTEGER
- [in] work\_id : INTEGER
- [out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num. You can get the work area number from the work area name using VT\_GET\_AREID.
- Specify the vision register number to store information on an allocated workpiece as vreg\_num.
- Specify time in milliseconds for which the robot waits for a workpiece as timeout. If no workpiece reaches the work area by timeout, the status becomes 117249. When a negative value is specified, the robot waits indefinitely.
- Usually, specify 1 as consecutive. To allocate workpieces successively regardless of the load balance to each work area, which is specified in the Line setup page, specify 2 or greater value. The detailed explanation is introduced in the Pick Program subsection in the *iRPickTool (Auto Visual Track Frame Setup) QUICK SETTING GUIDE (MARGGPTVT11171E)* or *iRPickTool (Auto Visual Track Frame Setup) QUICK SETTING GUIDE (B-83924EN-1)*.
- When workpiece information is to be allocated by specifying a particular model ID, specify the model ID as model\_id. Usually, this argument is not specified. If no model ID is specified, set an uninitialized variable (which has been declared but not stored with a value) to this argument. A use example is provided in the *iRPickTool (Auto Visual Track Frame Setup) QUICK SETTING GUIDE (MARGGPTVT11171E)* or *iRPickTool (Auto Visual Track Frame Setup) QUICK SETTING GUIDE (B-83924EN-1)*.
- When the ID of the workpiece to be allocated is known, specify it as work\_id. This ID is the unique identifier specified by VT\_PUT\_QUEUE. Usually, this argument is not specified. If no ID is specified, set an uninitialized variable to this argument.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.42 VT\_GET\_TIME iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It returns the estimated time to take until the next workpiece that can be handled arrives at a specified work area.

**Syntax:** VT\_GET\_TIME(area\_num, consecutive, model\_id, work\_id, time, status)

Input/Output Parameters:

[in] area\_num : INTEGER

[in] consecutive : INTEGER

[in] model\_id : INTEGER

[in] work\_id : INTEGER

[out] time : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num. You can get the work area number from the work area name using VT\_GET\_AREID.
- Usually, specify 1 as consecutive. If workpieces are to be allocated successively regardless of the load balance to each work area, which is specified in the Line setup page, and if you need the estimated time based on such allocation, specify 2 or greater value.
- When the estimated time of workpiece is to be gotten by specifying a particular model ID, specify the model ID as model\_id. Usually, this argument is not specified. If no model ID is specified, set an uninitialized variable to this argument.
- When the ID of the workpiece is known, specify it as work\_id. This ID is the unique identifier specified by VT\_PUT\_QUEUE. Usually, this argument is not specified. If no work ID is specified, set an uninitialized variable to this argument.
- The estimated time is stored in time. If the next workpiece that can be handled has not arrived at the work area, a positive value is stored in time. The unit is millisecond. For example, when 1000 is stored, the estimated time to take until the next workpiece arrives is about 1000 millisecond. If a workpiece that can be handled has already arrived at the work area, then a negative value is stored.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.43 VT\_GET\_TRYID iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It returns the tray number got from a specified tray name.

**Syntax:** VT\_GET\_TRYID(tray\_name)

Function Return Type: INTEGER

Input/Output Parameters:

[in] tray\_name : STRING

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a tray name as tray\_name.
- This built-in returns the tray number that is specified as an argument for VT\_PUT\_QUEUE.
- This built-in needs the *iRVision KAREL* interface option (J870) and the *iRVision Tracking Queue* option (J874), or PickTool + *iRVision* (R662), or PickTool – *iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.44 VT\_PUT\_QUE2 *iRVision* Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It pushes workpiece information into a tracking queue and outputs a duplication status that indicates whether the new part is pushed into the queue or not pushed because of double detection. The syntax of this built-in is almost the same as VT\_PUT\_QUEUE KAREL built-in except an argument to indicate the duplication status.

**Syntax:** VT\_PUT\_QUE2(line\_num, work\_id, tray\_num, enc\_count, model\_id, offset, found\_pos, meas\_val, duplicated, status)

Input/Output Parameters:

[in] line\_num : INTEGER  
 [in] work\_id : INTEGER  
 [in] tray\_num : INTEGER  
 [in] enc\_count: : INTEGER  
 [in] model\_id : INTEGER [in] offset: XYZWPR  
 [in] found\_pos : ARRAY[4] OF XYZWPR  
 [in] meas\_val : ARRAY[10] OF REAL  
 [out] duplicated : BOOLEAN  
 [out] status: INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- If a new pushed part already exists in a tracking queue, duplicated becomes TRUE and the part is not pushed into the queue. Otherwise, if the new pushed part does not exist in the queue, duplicated becomes FALSE and the part is pushed into the queue.
- The specifications of the other arguments are the same as VT\_PUT\_QUEUE.
- This built-in needs the *iRVision KAREL* interface option (J870) and the *iRVision Tracking Queue* option (J874), or PickTool + *iRVision* (R662), or PickTool – *iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.

**⚠ CAUTION**

This built-in cannot push a tray.

## A.22.45 VT\_PUT\_QUEUE iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It pushes workpiece information into a tracking queue.

**Syntax:** VT\_PUT\_QUEUE(line\_num, work\_id, tray\_num, enc\_count, model\_id, offset, found\_pos, meas\_val, status)

Input/Output Parameters:

[in] line\_num : INTEGER  
[in] work\_id : INTEGER  
[in] tray\_num : INTEGER  
[in] enc\_count : INTEGER  
[in] model\_id : INTEGER  
[in] offset : XYZWPR  
[in] found\_pos : ARRAY[4] OF XYZWPR  
[in] meas\_val : ARRAY[10] OF REAL  
[out] status : INTEGER  
%ENVIRONMENT Group : CVIS

**Details:**

- Specify a line number as line\_num. The workpiece information is pushed into the queue of the most upstream work area. You can get the line number from the line name using VT\_GET\_LINID.
- Specify the unique identifier of the workpiece as work\_id when needed. Please specify work\_id so that all the workpieces that are pushed into the queue have different identifiers when you control handling with the workpieces' IDs specified as the argument of VT\_GET\_QUEUE. Please specify 0 when you do not use the IDs.
- Specify a tray number as tray\_num. If tray is not used, specify 0. You can get the tray number from the tray name using VT\_GET\_TRYID.
- Specify an encoder value as enc\_count. It is the encoder value got at the moment when the image is acquired in order to find the workpiece or when the sensor such as photo eye detects the workpiece.
- Specify the model id for the workpiece as model\_id.
- Specify the offset data of the workpiece as offset.
- Specify the found positions as found\_pos[1-4].
- Specify the measurement values as meas\_val[1-10].
- Usually, just specify the values acquired by VT\_GET\_FOUND built-in from enc\_count to meas\_val.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.46 VT\_READ\_PQ iRVision Built-In Procedure

**Purpose:** This is a built-in for visual tracking customization. It reads the information of parts in a queue and copies them to a KAREL PATH variable.

**⚠ CAUTION**

This built-in cannot be used together with Load Balance function. This built-in cannot read the information of a cell of a tray by default. If you want to read the information of a cell of a tray, you set a flag to the system variable `$VTLINE[n].$FLAG` in all robots before you start the robot controller program. You will be able to read the information of a cell of a tray in V7.70P/30 and later.

1. Find the system variable `$VTLINE[n]` whose `$NAME` is identical to the name of the line where the work area belongs.
2. Divide `$VTLINE[n].$FLAG` by 128 and get the quotient of the division.
3. If the quotient is an even number, add 128 to `$VTLINE[n].$FLAG` to enable the function. If the quotient is an odd number, do nothing because the function has already been enabled.
4. This setting is required for all robots.

**Syntax:** `VT_READ_PQ(area_num, vtpartq, status)`

**Input/Output Parameters:**

[in] `area_num` : INTEGER

[out] `vtpartq` : PATH

[out] `status` : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as `area_num` in order to identify the queue of the work area. You can get the work area number from the work area name using `VT_GET_AREID`.
- Specify the KAREL PATH variable where the information of parts are copied from the queue.
- The type of the KAREL PATH variable is defined in advance in `VSTKTYPS.KL` that exists in the support folder which is created in the Robot folder of the ROBOGUIDE workcell by doing Build for a KAREL source. You cannot use another type for the PATH variable.

```
%ENVIRONMENT vstkdef
TYPE
    VTPARTQ_T = PATH NODEDATA = VT_PART_ND_T, PATHHEADER =
    VT_PART_HD_T
```

- `VTPARTQ_T` is the type of the PATH variable where the information of parts are copied from the queue. When you make a KAREL program, please make it include `VSTKTYPS.KL` and define the PATH variable.
- The header and node of the PATH variable are defined as follows.

```
TYPE
    VT_PART_HD_T = STRUCTURE
        area_id      : INTEGER
        num_parts   : INTEGER
    ENDSTRUCTURE
    VT_PART_ND_T = STRUCTURE
        work_id      : INTEGER
        model_id     : INTEGER
        tray_id      : INTEGER
        enc_count    : INTEGER
        offset       : VT_POS_T
        found_pos   : VT_POS_T
```

```

meas_value : ARRAY[10] OF REAL
ENDSTRUCTURE
VT_POS_T = STRUCTURE
  x : REAL
  y : REAL
  z : REAL
  w : REAL
  p : REAL
  r : REAL
ENDSTRUCTURE

```

### **Member of Header (VT\_PART\_HD\_T)**

#### **area\_id**

area id.

#### **num\_parts**

Number of parts that are copied from a queue to this PATH variable. Only the num\_parts nodes make sense. For example, if the total number of node is 100 and num\_parts is 10, the nodes from node 1 to node 10 make sense.

### **Member of Node (VT\_PART\_ND\_T)**

#### **work\_id**

Index number of the part. Sensor task assigns unique index numbers for each found part or tray.

#### **model\_id**

Model ID.

#### **tray\_id**

Index number of a tray. You can use this index number in order to identify tray information. The tray information are stored in \$VTTRAY[n] if \$STRAY\_ID = n. If tray is not used, this ID is uninitialized.

#### **enc\_count**

Encoder count. This shows the encoder value got at the moment when the image where the part has been found is snapped or when a sensor such as photo eye detects this part.

#### **offset**

Offset data represented based on tracking frame.

#### **found\_pos**

Found position represented based on tracking frame.

#### **meas\_value**

Measurement values set by the measurement output tool of the vision process which found the part.

- The smaller index number of node is, the more downstream the part is. That is, node 1 indicates the most downstream part in a queue.

#### **⚠ CAUTION**

If you read the information of a cell of a tray which uses sequence numbers, then it is not always true that the smaller index number of node is, the more downstream the part is.

- When a queue has parts more than the number of nodes, the information of excess parts is not indicated in these nodes. You can know the current total number of parts in a queue from `$VTAREASTAT[n]. $NUM_PARTS`.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the *iRVision KAREL* interface option (J870) and the *iRVision Tracking Queue* option (J874), or PickTool + *iRVision* (R662), or PickTool – *iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.
- You need the Virtual Robot with V7.70P/23 or later in order to translate this built-in.

**Example:**

- Make a PATH variable to restore the information of parts. The following KAREL program makes 200 nodes. Of course, you can also make nodes more than 200.

```
-----
PROGRAM makepq
-----
%COMMENT='sample'
%NOLOCKGROUP
%ENVIRONMENT pathop
%INCLUDE vstktyps
-----
VAR
-----
    vtpartq FROM makepq : VTPARTQ_T
    stat : INTEGER
    i     : INTEGER
-----
BEGIN
-----
    -- Make nodes for a queue
    For i = 1 TO 200 DO
        append_node(vtpartq, stat)
        IF stat <> 0 THEN
            ABORT
        ENDIF
    ENDFOR
END makepq
```

** CAUTION**

Please specify DRAM or CMOS for the PATH variable of part information. You cannot specify SHADOW for the PATH variable of part information. When you use DRAM, memory contents do not retain their stored values when it is turned off. But the processing speed is improved. On the other hand, when you use CMOS, memory contents retain their stored values when it is turned off. But the processing speed is slower. When there are a lot of parts, if you use CMOS, the processing speed may be very slow. In that case, you should use DRAM. Even if you use DRAM, you can save the DRAM variable contents to Flash ROM by using `SAVE_DRAM` built-in.

- When you want to access the PATH variable defined in another KAREL program as the above KAREL program, you need to change `VSTKTYPE.SKL` as follows. Otherwise, you cannot translate that KAREL program.

```
%ENVIRONMENT vstkdef
TYPE
```

```
VTPARTQ_T FROM makepq = PATH NODEDATA = VTPARTNODE_T,
PATHHEADER
= VTPARTHEADER_T
```

- Read the information of parts from a queue.

```
-----
PROGRAM readpq
-----
%COMMENT='sample'
%NOLOCKGROUP
%ENVIRONMENT cvis
%INCLUDE vstktyps
%INCLUDE klevccdf
%INCLUDE vierrdef
-----
CONST
-----
-- Arg Type
INT_TYPE = 1
REL_TYPE = 2
STR_TYPE = 3
-- Error Code
ER_SUCCESS = 0
-----
VAR
-----
vtpartq FROM makepq : VTPARTQ_T
area_name : STRING[20]
area_id : INTEGER
data_type : INTEGER
dmy_int : INTEGER
dmy_real : REAL
stat : INTEGER
severity : INTEGER
err_code : INTEGER
-----
BEGIN
-----
-- Get area name from argument
GET_TPE_PRM(1, data_type, dmy_int, dmy_real, area_name, stat)
IF stat <> ER_SUCCESS THEN
    WRITE TPERROR (CHR(cc_clear_win), CHR(cc_home))
    WRITE TPERROR ('Parameter missing (1: Area name)')
    ABORT
ENDIF
IF data_type <> STR_TYPE THEN
    WRITE TPERROR (CHR(cc_clear_win), CHR(cc_home))
    WRITE TPERROR ('Illegal parameter (1: Area name)')
    ABORT
ENDIF
-- Get area id
area_id = VT_GET_AREID(area_name)
IF area_id = -1 THEN
    -- NOT MATCH AREA
    WRITE TPERROR (CHR(cc_clear_win), CHR(cc_home))
    WRITE TPERROR ('Illegal parameter (1: area name)')
    ABORT
ENDIF
-- Read the information of parts
VT_READ_PQ(area_id, vtpartq, stat)
```

```

    IF (stat <> ER_SUCCESS) THEN
        severity = TRUNC(stat / 10000000)
        err_code = stat - severity * 10000000
        POST_ERR(err_code, '', 0, 1)
        ABORT
    ENDIF
END readpq

```

## A.22.47 VT\_SET\_FLAG iRVision Built-In Procedure

---

**Purpose:** This is a built-in for visual tracking. It enables or disables the multiple functions for a specified line or specified work areas at the same time.

**Syntax:** VT\_SET\_FLAG (line\_num, line\_bit2enb, line\_bit2dab, area\_bit2enb, area\_bit2dab, status)

Input/Output Parameters:

[in] line\_num : INTEGER

[in] line\_bit2enb : INTEGER

[in] line\_bit2dab : INTEGER

[in] area\_bit2enb : ARRAY[32] OF INTEGER

[in] area\_bit2dab : ARRAY[32] OF INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a line number as line\_num. You can get the line number from the line name using VT\_GET\_LINID.
- Specify the value (flag) that represents functions to be enabled for the line as line\_bit2enb. Correspondence between the specified values and the functions to be enabled is as below.
  - 1: [Load Balance] is enabled.
  - 2: [Allocated in ascending order] is enabled, and the workpieces are allocated in ascending order from the most downstream work area. The above multiple functions can be enabled at the same time. For example, if 3 (= 1 + 2) is specified, then [Load Balance] and [Allocated in ascending order] are enabled at the same time.
- Specify the value (flag) that represents functions to be disabled for the line as line\_bit2dab. Correspondence between the specified value and the function to be disabled is as below.
  - 1: [Load Balance] is disabled.
  - 2: [Allocated in ascending order] is disabled, and the workpieces are allocated in descending order from the most upstream work area.

The above multiple functions can be disabled at the same time. For example, if 3 (= 1 + 2) is specified, then [Load Balance] and [Allocated in ascending order] are disabled at the same time.

- Positive values can not be set both to line\_bit2enb and line\_bit2dab at the same time. Please set a positive value either to line\_bit2enb or line\_bit2dab, and set 0 to the other.
- Specify the values (flags) that represent functions to be enabled for work areas as area\_bit2enb. Use area\_bit2enb[1] for the most upstream work area. Use area\_bit2enb[2] for the second most upstream work area. Correspondence between the specified values as area\_bit2enb[n] and the functions to be enabled is as below.

- 1: [Skip this work area] is enabled.
- 2: [Y SORT] is enabled.
- 4: [Y SORT] is descending order.
- 8: [Stop/start Conveyor] is enabled.
- 16: [DO = OFF to stop] is selected.

The above multiple functions can be enabled at the same time. For example, if 3 (= 1 + 2) is specified, then [Skip this work area] and [Y SORT] are enabled at the same time.

- Specify the values (flags) that represent functions to be disabled for work areas as area\_bit2dab. Use area\_bit2dab[1] for the most upstream work area. Use area\_bit2dab[2] for the second most upstream work area. Correspondence between the specified values as area\_bit2dab[n] and the functions to be disabled is as below.

- 1: [Skip this work area] is disabled.
- 2: [Y Sort] is disabled.
- 4: [Y Sort] is ascending order.
- 8: [Stop/start Conveyor] is to be disabled.
- 16: [DO = ON to stop] is selected.

The above multiple functions can be disabled at the same time. For example, if 3 (= 1 + 2) is specified, then [Skip this work area] and [Y SORT] are disabled at the same time.

- Positive values can not be set both to area\_bit2enb[n] and area\_bit2dab[n] at the same time. Please set a positive value either to area\_bit2enb[n] or area\_bit2dab[n], and set 0 to the other.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the *iRVision KAREL* interface option (J870) and the *iRVision Tracking Queue* option (J874), or PickTool + *iRVision* (R662), or PickTool – *iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.48 VT\_SET\_LDBAL *iRVision* Built-In Procedure

**Purpose:** This is a built-in for visual tracking. It changes the load balance data of each work area on a specified line.

**Syntax:** VT\_SET\_LDBAL (line\_num, model\_id, n2pick, n2pass, status)

Input/Output Parameters:

[in] line\_num : INTEGER

[in] model\_id : INTEGER

[in] n2pick : ARRAY[32] OF INTEGER

[in] n2pass : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : CVIS

**Details:**

- Specify a line number as line\_num. You can get the line number from the line name using VT\_GET\_LINID.
- Specify a model ID of workpieces for which the load balance should be changed as model\_id. When [Common for all model IDs] is selected, please specify 0.

- Specify rate of workpieces handled in each work area as n2pick. Specify the rate of workpieces handled in the most upstream work area in the specified line as n2pick[1]. Specify the rate of workpieces handled in the second most upstream work area as n2pick[2].
- Specify the rate of [Bypass] as n2pass.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the *iRVision KAREL* interface option (J870) and the *iRVision Tracking Queue* option (J874), or PickTool + *iRVision* (R662), or PickTool – *iRVision* (R718).
- This built-in is not available in controller software versions V8.20 and higher.

## A.22.49 VT\_WRITE\_PQ *iRVision* Built-In Procedure

**Purpose:** This is a built-in for visual tracking customization. It writes the information of a part in a queue. The part is identified by a work ID.

 **CAUTION**

This built-in cannot be used together with Load Balance function. This built-in cannot write the information of a cell of a tray by default. If you want to write the information of a cell of a tray, you set a flag to the system variable `$VTLINE[n].$FLAG` in all robots before you start the robot controller program. You will be able to write the information of a cell of a tray in V7.70P/30 and later.

1. Find the system variable `$VTLINE[n]` whose `$NAME` is identical to the name of the line where the work area belongs.
2. Divide `$VTLINE[n].$FLAG` by 128 and get the quotient of the division.
3. If the quotient is an even number, add 128 to `$VTLINE[n].$FLAG` to enable the function. If the quotient is an odd number, do nothing because the function has already been enabled.
4. This setting is required for all robots.

**Syntax:** `VT_WRITE_PQ(area_num, work_id, model_id, enc_count, offset, found_pos, meas_value, status)`

Input/Output Parameters:

[in] `area_num` : INTEGER  
 [in] `work_id` : INTEGER  
 [in] `model_id` : INTEGER  
 [in] `enc_count` : INTEGER  
 [in] `offset` : XYZWPR  
 [in] `found_pos` : XYZWPR  
 [in] `meas_value` : ARRAY[10] OF REAL  
 [out] `status` : INTEGER  
 %ENVIRONMENT Group : CVIS

**Details:**

- Specify a work area number as area\_num in order to identify the queue of the work area. You can get the work area number from the work area name using VT\_GET\_AREID.
- Specify the index number of a part of the queue as work\_id. You can know the index number by reading the PATH variable's nodes gotten by VT\_READ\_PQ. Sensor task assigns unique index numbers for each found part.
- Specify new values from model\_id to meas\_value. You can write the following information. When you do not want to change some of the existing values, you specify uninitialized values as them.
  - Model ID. For example, you can assign each robot to handle parts with different model IDs.
  - Encoder count. This shows the encoder value got at the moment when the image where the part has been found is snapped or when a sensor such as photo eye detects this part.
  - Offset data represented by tracking frame.
  - Found positions represented by tracking frame.
  - Measurement values set by the measurement output tool of the vision process which found the part. On the other hand, if you use a third-party sensor, you can add the result of that sensor to the measurement values.
- The status of the process is returned to status. If the process is successful, then status is set to 0. If the process is not successful, then an alarm code that is not equal to 0 is returned.
- This built-in needs the iRVision KAREL interface option (J870) and the iRVision Tracking Queue option (J874), or PickTool + iRVision (R662), or PickTool – iRVision (R718).
- This built-in is not available in controller software versions V8.20 and higher.
- You need the Virtual Robot with V7.70P/23 or later in order to translate this built-in.

**Example:**

- Write the information of a part in a queue.

```

-----  

PROGRAM writepq  

-----  

%COMMENT='sample'  

%NOLOCKGROUP  

%ENVIRONMENT cvis  

%INCLUDE vstktyps  

%INCLUDE vierrdef  

-----  

CONST  

-----  

-- Error Code  

ER_SUCCESS = 0  

-----  

VAR  

-----  

    vtpartq FROM makepq : VTPARTQ_T  

    area_id      : INTEGER  

    work_id      : INTEGER  

    model_id     : INTEGER  

    enc_cnt      : INTEGER  

    offset       : XYZWPR  

    found_pos    : XYZWPR  

    meas_value   : ARRAY[10] OF REAL  

    stat         : INTEGER  

    severity     : INTEGER  

    err_code     : INTEGER  

    i             : INTEGER  

-----  

BEGIN

```

```

-----  

FOR i = 1 TO vtpartq.num_parts DO
    -- Select the specific part
    if vtpartq[i].model_id = 1 THEN
        -- Change model ID for the specific part
        model_id = vtpartq[i].model_id + 100
        -- Write model ID for the specific part
        area_id = vtpartq.area_id
        work_id = vtpartq[i].work_id
        VT_WRITE_PQ(area_id, work_id, enc_cnt, model_id, offset,
        found_pos, meas_value, stat)
        IF (stat <> ER_SUCCESS) THEN
            severity = TRUNC(stat / 10000000)
            err_code = stat - severity * 10000000
            POST_ERR(err_code, '', 0, 1)
            ABORT
        ENDIF
    ENDIF
ENDFOR
END writepq

```

## A.23 - W - KAREL LANGUAGE DESCRIPTION

---

### A.23.1 WAIT FOR Statement

---

**Purpose:** Delays continuation of program execution until some condition(s) are met

**Syntax:** WAIT FOR cond\_list

where:

cond\_list : one or more conditions

**Details:**

- All of the conditions in a single WAIT FOR statement must be satisfied simultaneously for execution to continue.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

### A.23.2 WHEN Clause

---

**Purpose:** Used to specify a conditions/actions pair in a global condition handler

**Syntax:** WHEN cond\_list DO action\_list

where:

cond\_list : one or more conditions

action\_list : one or more conditions

**Details:**

- All of the conditions in the cond\_list of a single WHEN clause must be satisfied simultaneously for the condition handler to be triggered.
- The action\_list represents a list of actions to be taken when the corresponding conditions of a WHEN clause are satisfied simultaneously.
- Calls to function routines are not allowed in a CONDITION statement and, therefore, cannot be used in a WHEN clause.
- CONDITION statements can include multiple WHEN clauses.

**See Also:** [Chapter 6, CONDITION HANDLERS](#) , [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to the following sections for detailed program examples:

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.9, USING DYNAMIC DISPLAY BUILT-INS \(DYN\\_DISP.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

### A.23.3 WHILE...ENDWHILE Statement

**Purpose:** Used when an action is to be executed as long as a BOOLEAN expression remains TRUE

**Syntax:** WHILE boolean\_exp DO{ statement }ENDWHILE

where:

boolean\_exp : a BOOLEAN expression

statement : a valid KAREL executable statement

**Details:**

- boolean\_exp is evaluated before each iteration.
- As long as boolean\_exp is TRUE, the statements in the loop are executed.
- If boolean\_exp is FALSE, control is transferred to the statement following ENDWHILE, and the statement or statements in the body of the loop are not executed.

**See Also:** [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to [Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#) for a detailed program example.

### A.23.4 WITH Clause

**Purpose:** Used in condition handlers to specify condition handler qualifiers

**Syntax:** WITH param\_spec {, param\_spec}

where:

param\_spec is of the form : with\_sys\_var = value

with\_sys\_var : one of the system variables available for use in the WITH clause

value : an expression of the type corresponding to the type of the system variable

#### Details:

- The actual system variables specified are not changed.
- \$PRIORITY and \$SCAN\_TIME are condition handler qualifiers that can be used in a WITH clause only when the WITH clause is part of a condition handler statement.

## A.23.5 WRITE Statement

---

**Purpose:** Writes data to a serial device or file

**Syntax:** WRITE <file\_var> (data\_item { ,data\_item })

where:

file\_var : a FILE variable

data\_item : an expression and its optional format specifiers or the reserved word CR

#### Details:

- If file\_var is not specified in a WRITE statement the default TPDISPLAY is used. %CRTDEVICE directive will change the default to OUTPUT.
- If file\_var is specified, it must be one of the output devices or a variable that has been equated to one of them.
- If file\_var attribute was set with the UF option, data is transmitted to the specified file or device in binary form. Otherwise, data is transmitted as ASCII text.
- data\_item can be any valid KAREL expression.
- If data\_item is of type ARRAY, a subscript must be provided.
- If data\_item is of type PATH, you can specify that the entire path be read, a specific node be read [n], or a range of nodes be read [n .. m].
- Optional format specifiers can be used to control the amount of data that is written for each data\_item.
- The reserved word CR, which can be used as a data item, specifies that the next data item to be written to the file\_var will start on the next line.
- Use the IO\_STATUS built-in to determine if the write operation was successful.

**See Also:** [Section A.16.1, PATH Data Type](#), for more information on writing PATH variables, [Chapter 7, FILE INPUT/OUTPUT OPERATIONS](#), for more information on format specifiers and file\_vars. [Appendix E, SYNTAX DIAGRAMS](#) for more syntax information

**Example:** Refer to [Appendix B, KAREL EXAMPLE PROGRAMS](#) for more detailed program examples.

## A.23.6 WRITE\_DICT Built-In Procedure

---

**Purpose:** Writes information from a dictionary

**Syntax:** WRITE\_DICT(file\_var, dict\_name, element\_no, status)

Input/Output Parameters:

[in] file\_var : FILE  
[in] dict\_name : STRING  
[in] element\_no : INTEGER  
[out] status : INTEGER  
%ENVIRONMENT Group : PBCORE

**Details:**

- file\_var must be opened to the window where the dictionary text is to appear.
- dict\_name specifies the name of the dictionary from which to write.
- element\_no specifies the element number to write. This number is designated with a \$ in the dictionary file.
- status explains the status of the attempted operation. If not equal to 0, then an error occurred writing the element from the dictionary file.

**See Also:** [Section A.18.2, READ\\_DICT Built-In Procedure](#) , [Section A.18.9, REMOVE\\_DICT Built-In Procedure](#) , [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** Refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE \(DCLST\\_EX.KL\)](#), for a detailed program example.

## A.23.7 WRITE\_DICT\_V Built-In Procedure

**Purpose:** Writes information from a dictionary with formatted variables

**Syntax:** WRITE\_DICT\_V(file\_var, dict\_name, element\_no, value\_array, status)

Input/Output Parameters:

[in] file\_var : FILE  
[in] dict\_name : STRING  
[in] element\_no : INTEGER  
[in] value\_array : ARRAY OF STRING  
[out] status : INTEGER  
%ENVIRONMENT Group : UIF

**Details:**

- file\_var must be opened to the window where the dictionary text is to appear.
- dict\_name specifies the name of the dictionary from which to write.
- element\_no specifies the element number to write. This number is designated with a \$ in the dictionary file.
- value\_array is an array of variable names that corresponds to each formatted data item in the dictionary text. Each variable name may be specified as '[prog\_name] var\_name'.
  - [prog\_name] specifies the name of the program that contains the specified variable. If not specified, then the current program being executed is used.
  - var\_name must refer to a static, global program variable.
  - var\_name may contain node numbers, field names, and subscripts.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred writing the element from the dictionary file.

**See Also:** [Section A.18.3, READ\\_DICT\\_V Built-In-Procedure](#) , [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:** In the following example, TPTASKEG.TX contains dictionary text information which will display a system variable. This information is the first element in the dictionary and element numbers start at 0. util\_prog uses WRITE\_DICT\_V to display the text on the teach pendant.

```
-----
TPTASKEG.TX
-----
$ "Maximum number of tasks = %d"
-----
UTILITY PROGRAM:
-----
PROGRAM util_prog
%ENVIRONMENT uif
VAR
    status: INTEGER
    value_array: ARRAY[1] OF STRING[30]
BEGIN
    value_array[1] = '[*system*].$scr.$maxnumtask'
    ADD_DICT('TPTASKEG', 'TASK', dp_default, dp_open, status)
    WRITE_DICT_V(TPDISPLAY, 'TASK', 0, value_array, status)
END util_prog
```

**Figure A.23.7 WRITE\_DICT\_V Built-In Procedure**

## A.24 - X - KAREL LANGUAGE DESCRIPTION

### A.24.1 XML\_ADDTAG Built-In Procedure

**Purpose:** Associates the tag tag\_name with the xml\_file.

**Syntax:** XML\_ADDTAG(xml\_file, tag\_name, numchar, caseflag, tag\_ident, status)

Input/Output Parameters:

[in] xml\_file : FILE

[in] tag\_name : STRING

[in] numchar: INTEGER

[in] caseflag : BOOLEAN

[in] tag\_ident : INTEGER

[out] status : INTEGER

%ENVIRONMENT Group : PBCORE

**Details:**

- xml\_file specifies an open KAREL file with AR\_XML attribute set

- tag\_name is the name of the tag you want to know about
- numchar specifies the number of characters to use when looking for the tag
- caseflag – If TRUE, specifies whether the tag\_name is case-sensistive
- tag\_ident – Application identifier that user associates with tag. There are some system tag idents that the user cannot use. This is used to allow you to switch on his tags and the user return codes. When the scanning encounters the registered tag it will return with the tag\_ident. The file MUST be open before you can register a tag. tag\_ident should be a unique number within the application. This allows the application to do a select based on the return identifier.

**NOTE**

The system reserves some identifiers for error and scan limit status returns. This allows the application to easily include these constants in the SELECT statement.

- status explains the status of the attempted operation. If not equal to 0, then an error occurred writing the element from the dictionary file. The return status will be bad if the user has not opened the file and set the XML attribute.

**Example:** Refer to [Section 10.4, FORMATTING XML INPUT](#).

## **A.24.2 XML\_GETDATA Built-In Procedure**

**Purpose:** Returns the attribute names and values associated with the tag causing the return.

**Syntax:** XML\_GETDATA(xml\_file, numattr, attrnames, attrvalues, text, textdone)

Input/Output Parameters:

[in] xml\_file : FILE  
[out] numattr : INTEGER  
[out] attrnames : ARRAY OF STRING  
[out] attrvalues : ARRAY OF STRING  
[out] textdata : STRING  
[out] textdone : BOOLEAN  
[out] status : INTEGER

**Details:**

- xml\_file is an open KAREL file with AR\_XML attribute set
- numattr indicates the number of attributes
- attrnames indicates attribute names
- attrvalues indicates attribute values
- textdata indicates the text that follows the tag
- textdone – If this is FALSE, more text is to be read
- status indicates the result of the operation

## A.24.3 XML\_REMTAG Built-In Procedure

---

**Purpose:** Removes the tag name from the list.

**Syntax:** XML\_REMTAG(xml\_file, tag\_name, tag\_ident, status)

Input/Output Parameters:

[in] xml\_file : FILE

[in] tag\_name : STRING

[OUT] status : INTEGER

**Details:**

- xml\_file – Open KAREL file with AR\_XML attribute set
- tag\_name – Indicates the name of the tag to remove
- status – Indicates the result of the operation

 **CAUTION**

Do not attempt to remove a tag while it is in use.

## A.24.4 XML\_SCAN Built-In Procedure

---

**Purpose:** Scan through a previously opened XML file

**Syntax:** XML\_SCAN(xml\_file, tag\_name, func\_code, status)

Input/Output Parameters:

[in] xml\_file : FILE

[out] tag\_name: STRING

[out] tag\_ident : INTEGER

[out] func\_code : INTEGER

[out] status : INTEGER

**Details:**

- xml\_file – Open KAREL file with AR\_XML attribute set
- tag\_name – Name of the tag system found
- tag\_ident – Tag user associated in addtag call
- func\_code – Function code, start and so forth
- status – Result of operation
- When a registered tag is found
  - func\_code = XML\_START or XML\_END or XML\_STEND
  - tag\_ident = The value associated with the tag when it was registered
- When a text buffer is full
  - func\_code = XML\_TXCONT
  - tag\_ident = The value associated with the tag when it was registered

- After 50 lines are scanned
  - status = XML\_SCANLIM, just recall the built-in when the is encountered
  - tag\_ident = XML\_SCANLIM
  - This is not an error and indicates that there is more to come
- If it encounters a parsing error
  - status = Some error
  - tag\_ident = XML\_ERROR
- At the end of file, status = SUCCESS
- Valid Parse Errors are:
  - XML\_TAG\_SIZE – Too many characters in tag
  - XML\_ATTR\_SIZE – Too many characters in attribute
  - XML\_NOSLASH – Invalid use of / character
  - XML\_INVTAG – Invalid character in tag
  - XML\_UNMATCHATTR – No value for attribute
  - XML\_UNMATCHTAG – End tag with no matching start
  - XML\_INVATTR – Invalid character in attribute
  - XML\_NOFILE – Cannot find file
  - XML\_TAGNEST – Tag nesting level too deep
  - XML\_COMMENT – Error in comment
- The system will provide a separate return for the start of the tag and the end of the tag if the tag does not contain both starting and ending information. The attribute data is NOT valid when the call is made for the END tag.
- The tag\_ident will be the tag\_ident that the user registered for the registered tag return. If the system returns for other reasons then the tag\_ident may contain system tag data.

## A.24.5 XML\_SETVAR Built-In Procedure

**Purpose:** Sets the variable [prog\_name]var\_name according to the attributes that were associated with the tag causing the return.

**Syntax:** XML\_SETVAR(xml\_file, prog\_name, var\_name, status)

Input/Output Parameters:

[in] xml\_file : FILE  
[in] prog\_name : STRING  
[in] var\_name : STRING  
[out] status : INTEGER

**Details:**

- In this case the text for the attribute will be matched with the text field name of the KAREL variable.  
So a variable of this type:

```
xmlstrct_t = STRUCTURE
    first: integer
    second: real
    third: BOOLEAN
```

```

fourth: string[20]
ENDSTRUCTURE

```

Can be set via the XML:

```

<xmlstrct_t first="123456" second="7.8910" third="1" fourth="A
string"\>

```

- The XML tag name does not need to match the TYPE name. The association of the field names and attribute names is based on the [program] variable in the call to XML\_SETVAR.

## A.24.6 XYZWPR Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as XYZWPR data type

**Syntax:** XYZWPR <IN GROUP [n]>

**Details:**

- An XYZWPR consists of three REAL components specifying a Cartesian location (x,y,z), three REAL components specifying an orientation (w,p,r), and a component specifying a CONFIG Data Type, 36 bytes total.
- The configuration string indicates the joint placements and multiple turns that describe the configuration of the robot when it is at a particular position.
- A position is always referenced with respect to a specific coordinate frame.
- Components of XYZWPR variables can be accessed or set as if they were defined as follows:

```

XYZWPR = STRUCTURE
  X: REAL
  Y: REAL
  Z: REAL
  W: REAL
  P: REAL
  R: REAL
  CONFIG_DATA: CONFIG
ENDSTRUCTURE
Note: All fields are read-write access.

```

**Figure A.24.6 XYZWPR Data Type**

**Example:** Refer to the following sections for detailed program examples:

[Section B.2, COPYING PATH VARIABLES \(CPY\\_PTH.KL\)](#)

[Section B.5, USING REGISTER BUILT-INS \(REG\\_EX.KL\)](#)

[Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM \(PTH\\_MOVE.KL\)](#)

[Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING \(DOUT\\_EX.KL\)](#)

## A.24.7 XYZWPREXT Data Type

**Purpose:** Defines a variable, function return type, or routine parameter as an XYZWPREXT

**Syntax:** XYZWPREXT <IN GROUP [n]>

**Details:**

- An XYZWPREXT consists of three REAL components specifying a Cartesian location (x,y,z), three REAL components specifying an orientation (w,p,r), and a component specifying a configuration string. It also includes three extended axes, 48 bytes total.
- The configuration string indicates the joint placements and multiple turns that describe the configuration of the robot when it is at a particular position.
- A position is always referenced with respect to a specific coordinate frame.
- Components of XYZWPREXT variables can be accessed or set as if they were defined as follows:

```
XYZWPRext = STRUCTURE
    X: REAL
    Y: REAL
    Z: REAL
    W: REAL
    P: REAL
    R: REAL
    CONFIG_DATA: CONFIG
    EXT1: REAL
    EXT2: REAL
    EXT3: REAL
ENDSTRUCTURE
--Note: All fields are read-write access.
```

Figure A.24.7 XYZWPREXT Data Type

## A.25 - Y - KAREL LANGUAGE DESCRIPTION

There are no KAREL descriptions beginning with Y.

## A.26 - Z - KAREL LANGUAGE DESCRIPTION

There are no KAREL descriptions beginning with Z.

## B KAREL EXAMPLE PROGRAMS

---

This appendix contains some KAREL program examples. These programs are meant to show you how to use the KAREL built-ins and commands described in [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#)

This section includes examples of how to use the KAREL built-ins and commands in a program. Refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more detailed information on each of the KAREL built-ins and commands.

[Table B](#) lists the programs in this section, their main function, the built-ins used in each program, and the section to refer to for the program listing.

### Conventions

Each program in this appendix is divided into five sections.

Section 0 - Lists each element of the KAREL language that is used in the example program.

Section 1 - Contains the program and environment declarations.

Section 2 - Contains the constant, variable, and type declarations.

Section 3 - Contains the routine declarations.

Section 4 - Contains the main body of the program.

**Table B KAREL Example Programs**

Program Name	Program Function	Built-ins Used	Section to Refer
CPY_PTH.KL	Copies path variables.	APPEND_NODE BY_NAME CALL_PROG CNV_INT_STR COPY_PATH CREATE_VAR CURPOS DELETE_NODE LOAD PATH_LEN PROG_LIST READ_KB SET_CURSOR SET_FILE_ATR SET_VAR SUB_STR VAR_LIST	<a href="#">Section B.2, COPYING PATH VARIABLES</a>

<b>Program Name</b>	<b>Program Function</b>	<b>Built-ins Used</b>	<b>Section to Refer</b>
SAVE_VRS.KL	Saves data to the default device.	DELETE_FILE SAVE	Section B.3, SAVING DATA TO THE DEFAULT DEVICE
ROUT_EX.KL	Contains standard routines that are used throughout the program examples.	CHR FORCE_SPMENU	Section B.4, STANDARD ROUTINES
REG_EX.KL	Uses Register built-ins.	CALL_PROGLIN CHR CURPOS GET_JPOS_REG GET_POS_REG GET_REG POS_REG_TYP SET_INT_REG SET_JPOS_REG SET_POS_REG FORCE_SPMENU	Section B.5, USING REGISTER BUILT-INS
PTH_MOVE.KL	Teaches and moves along a path. Also uses condition handlers.	CHR CNV_REL_JPOS PATH_LEN SET_CURSOR	Section B.6, POSITION DATA SET AND CONDITION HANDLERS PROGRAM
LIST_EX.KL	Lists files and programs, and manipulate strings.	ABS ARRAY_LEN CNV_INT_STR FILE_LIST LOAD LOAD_STATUS PROG_LIST ROUND SUB_STR	Section B.7, LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS
FILE_EX.KL	Uses the File and Device built-ins.	CNV_TIME_STR COPY_FILE DISMOUNT_DEV FORMAT_DEV GET_TIME MOUNT_DEV SUB_STR	Section B.8, USING THE FILE AND DEVICE BUILT-INS

Program Name	Program Function	Built-ins Used	Section to Refer
DYN_DISP.KL	Uses Dynamic Display built-ins.	ABORT_TASK CNC_DYN_DISB CNC_DYN_DISE CNC_DYN_DISP CNC_DYN_DISS CNC_DYN_DISI CNC_DYN_DISR INI_DYN_DISB INI_DYN_DISE INI_DYN_DISP INI_DYN_DISS INI_DYN_DISI INI_DYN_DISR LOAD LOAD_STATUS RUN_TASK	<a href="#">Section B.9, USING DYNAMIC DISPLAY BUILT-INS</a>
CHG_DATA.KL	Processes and changes values of dynamically displayed variables.		<a href="#">Section B.10, MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES</a>
DCLST_EX.KL	Displays a list from a dictionary file.	ADD_DICT ACT_SCREEN ATT_WINDOW_S CHECK_DICT CLR_IO_STAT CNV_STR_INT DEF_SCREEN DISCTRL_LIST FORCE_SPENU IO_STATUS ORD READ_DICT REMOVE_DICT SET_FILE_ATR SET_WINDOW STR_LEN UNINIT WRITE_DICT	<a href="#">Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE</a>
DCLISTEG.UTX	Dictionary file.	N/A	<a href="#">Section B.11.1, Dictionary Files</a>

Program Name	Program Function	Built-ins Used	Section to Refer
DCALP_EX.KL	Uses the DISCTRL_ALPHA Built-in.	ADD_DICT CHR DISCTRL_ALPH FORCE_SPEMU POST_ERR SET_CURSOR SET_LANG	<a href="#">Section B.12, USING THE DISCTRL_ALPHA BUILT-IN</a>
DCALPHEG.UTX	Dictionary file.	N/A	<a href="#">Section B.12.1, Dictionary Files</a>
CPY_TP.KL	Applies offsets to copied teach pendant programs.	AVL_POS_NUM CHR CLOSE_TPE CNV_JPOS_REL CNV_REL_JPOS COPY_TPE GET_JPOS_TYP GET_POS_TPE GET_POS_TYP OPEN_TPE PROG_LIST SELECT_TPE SET_JPOS_TPE SET_POS_TPE	<a href="#">Section B.13, APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM</a>
DOUT_EX.KL	Sets up digital output ports for process monitoring. The DOUTs are used to monitor the status of the external equipment and to show the current status of the process. The equipment status DOUTs are simulated, but in practice they are looked up to the actual external equipment as a feedback response. The robot is moved along a path until the external equipment needs servicing, which is triggered by the DOUT values.	CHRPATH_LEN CURPOS DELAY FORCE_SPMENU RESET SET_PORT_ASG SET_PORT_CMT SET_PORT_MOD SET_PORT_SIM	<a href="#">Section B.1, SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING</a>

Program Name	Program Function	Built-ins Used	Section to Refer
VSBLTIN.KL	A server side sample of a custom iRVision screen using V_CSAPI_*.	V_CSAPI_GETVALUE V_CSAPI_NUMSET V_CSAPI_RESETDATA V_CSAPI_SAVEDATA V_CSAPI_SETVALUE V_CSAPI_TESTRUN	<a href="#">Section B.14, iRVISION CUSTOM SCREEN USING V_CSAPI BUILT-INS</a>

## B.1 SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING

This program sets up digital output ports for process monitoring. The DOUTs are to monitor the external equipment status and show the current status of the process. The equipment status DOUTs are simulated, but in practice are hooked up to the actual external equipments as a feedback response. The robot is moved along a path until external equipment needs to be serviced, which is triggered by the DOUT values.

```
-----
----- DOUT_EX.KL
-----
----- Section 0: Detail about DOUT_EX.kl
-----
----- Elements of KAREL Language Covered:
----- Action:
-----     CONTINUE                               Sec 4-A
-----     ENABLE CONDITION                      Sec 3-B; 4-C
-----     NOMESSAGE                            Sec 4-A
-----     RESUME                                Sec 4-C
-----     ROUTINE CALL                          Sec 4-A,C
-----     SIGNAL EVENT                          Sec 4-C
-----     STOP                                  Sec 4-C
-----     UNPAUSE                               Sec 4-A
----- Clauses:
-----     FROM                                  Sec 3-A
-----     WHEN                                  Sec 4-A,C
----- Conditions:
-----     ERROR[xxx]                           Sec 4-A
-----     EVENT                                Sec 4-C
-----     RELATIONAL condition                  Sec 4-A
----- Data types:
-----     BOOLEAN                             Sec 2
-----     INTEGER                             Sec 2
-----     PATH                                Sec 2
-----     XYZWPREXT                          Sec 2
-----     STRING                               Sec 2
-----     XYZWPR                             Sec 2
----- Directives:
-----     ALPHABETIZE                         Sec 1
-----     COMMENT                             Sec 1
-----     CMOSVARS                           Sec 1
```

-----	INCLUDE	Sec 1
-----	Built-in Functions & Procedures:	
-----	CHR	Sec 3-E; 4-D
-----	CURPOS	Sec 3-E
-----	DELAY	Sec 3-B, E
-----	FORCE_SPMENU	Sec 3-E; 4-D
-----	PATH_LEN	Sec 4-B; 4-D
-----	RESET	Sec 3-B
-----	SET_PORT_ASG	Sec 3-D
-----	SET_PORT_CMT	Sec 3-D
-----	SET_PORT_MOD	Sec 3-C
-----	SET_PORT_SIM	Sec 4-D
-----	SET_POS_REG	Sec 3-E, 4-D
-----	SET_EPOS_REG	Sec 3-E
-----	Statements:	
-----	ABORT	Sec 3-D; 4-B, D
-----	ATTACH	Sec 4-B
-----	CONNECT TIMER	Sec 4-A
-----	CONDITION...ENDCONDITON	Sec 4-A, C
-----	ENABLE CONDITION	Sec 3-B, E; 4-A, C
-----	FOR...ENDFOR	Sec 3-D
-----	IF...THEN...ENDIF	Sec 3-B, C, D; 4-B, C, D
-----	RELEASE	Sec 4-B
-----	ROUTINE	Sec 3-A, B, C, D, E, F
-----	WAIT FOR	Sec 3-E
-----	WHILE...ENDWHILE	Sec 4-B
-----	WRITE	Sec 3-B, D, E; 4-B, D
-----	Reserve Words:	
-----	BEGIN	Sec 3-B, C, D, E; 4
-----	CONST	Sec 2
-----	CR	Sec 3-B, D, E; 4-B, D
-----	END	Sec 3-B, C, D, E, 4-D
-----	NOT	Sec 3-B; 4-C
-----	PROGRAM	Sec 1
-----	VAR	Sec 2
-----	Predefined FILE names:	
-----	TPFUNC	Sec 4-D

**Figure B.1 (a) Setting Up Digital Output Ports for Process Monitoring - Overview**

```
-----  
-----  
----- Section 1: Program and Environment Declaration  
-----  
  
PROGRAM DOUT_EX  
%ALPHABETIZE  
%NOPAUSE = TPENABLE  
%COMMENT = 'PORT/CH DOUT_EX'  
%CMOSVARS  
%INCLUDE KLIOTYPS  
-----  
-----  
----- Section 2: Constant and Variable Declarations  
-----  
  
CONST  
-- Condition Handler Numbers  
CONT_CH = 2  
EQIP_FAIL = 3  
RESTART = 6  
SERV_DONE = 4  
-- Continue execution condition  
-- Equipment Failure Condition  
-- Restart condition Handler  
-- Servicing Done condition
```

```

UNINIT_CH    = 10                      -- Monitor for uninit error
WARMED_UP    = 5                       -- Event to notify eqip is ready
-- Process DOUT numbers ( 1 thru 6 are complementary DOUT )
--                               ( 3 and 4 are simulated DOUT )
EQIP_READY   = 1                       -- Equipment Ready
EQIP_NOT_RD  = 2                       -- Equipment Not Ready
EQIP_ERROR   = 3                       -- Equipment Failed during process
EQIP_FIXED   = 4                       -- Equipment Fixed after failure
EQIP_ON      = 5                       -- Turn Eqip-1 ON DOUT
EQIP_OFF     = 6                       -- Turn Eqip-1 OFF DOUT
NODE_PULSE   = 7                       -- Node Pulsing DOUT
FINISH       = 8                       -- Path Finishing signal DOUT
-- Process Constants
SUCCESS      = 0                       -- Successful Operation Status
UNASIGNED    = 13007                   -- Unassigned Port Deletion Error
VAR
cont_timer,
last_node, node_ind,
status          :INTEGER           -- Status from built-in calls
prg_abrt       :BOOLEAN            -- Set when the program is aborted
pth1           :PATH               -- Process Path
stop_pos        :XYZWPREXT        -- Process Stop Position
perch_pos       :XYZWPR             -- Perch Position
tmp_xyz         :XYZWPR             -- XYZWPR variable for temporary
                                     -- use
indx           :INTEGER            -- Used a FOR loop counter
ports_ready    :BOOLEAN            -- Check if ports assigned
cmt_str        :STRING[10]          -- Comment String

```

**Figure B.1 (b) Setting Up Digital Output Ports for Process Monitoring - Declaration Section**

```

-----  

----- Section 3: Routine Declaration  

-----  

----- Section 3-A: TP_CLS Declaration  

----- This routine is from ROUT_EX.kl and will  

----- clear the TP USER menu screen and force  

----- it to be visible.  

-----  

ROUTINE tp_cls FROM rout_ex      -- ROUT_EX must also be loaded.  

-----  

----- Section 3-B: port_init Declaration  

----- This routine assigns a value to ports_ready,  

----- which allows the ports to be initialized.  

----- It resets the controller so that program  

----- execution may be continued automatically  

----- through the CONT_CH condition handler.  

-----  

ROUTINE init_port  

VAR
    reset_ok: BOOLEAN
BEGIN
    ports_ready = FALSE -- Set false so ports will be initialized
    DELAY 500;
    RESET(reset_ok)           -- Reset the controller
    IF (NOT reset_ok) THEN
        WRITE('Reset Failed', CR)
    ENDIF

```

```

cont_timer = 0          -- Set a timer to continue the process
ENABLE CONDITION[CONT_CH] -- Enabled the CONT_CH which continues
                           -- program execution
END init_port

-----
----- Section 3-C: SET_MODE Declaration
----- Sets up the mode of IO's. Depending on the
----- passed parameter the IO ports will be set to
----- REVERSE and/or COMPLEMENTARY mode. When the
----- ports are set to REVERSE mode, the TRUE
----- condition is represented by a FALSE signal.
----- When COMPLEMENTARY mode is selected for a
----- port (odd number port), the port n and n+1
----- are complementary signal of each other.
-----

ROUTINE set_mode(port_type:      INTEGER;
                  port_no:        INTEGER;
                  reverse:        BOOLEAN;
                  complmnt:       BOOLEAN)
VAR
    mode:              INTEGER
BEGIN -- set_mode
    IF reverse THEN
        mode = 1                      -- Set the reverse mode
    ELSE
        mode = 0
    ENDIF
    IF complmnt THEN
        mode = mode OR 2            -- Set complementary mode
    ENDIF
    SET_PORT_MOD(port_type, port_no, mode, status)
END set_mode

-----
----- Section 3-D: SETUP_PORTS Declaration
----- This section assumes that you do not have an AB or
----- GENIUS I/O or any other external I/O board.
----- Therefore, any previous port assignments are no
----- longer needed for this application, and
----- can be deleted.
-----

ROUTINE setup_ports
VAR
    port_n : INTEGER
BEGIN
-- Delete DIGITAL OUTPUT PORTS 1 thru 48
    FOR port_n = 0 to 5 DO
        -- Indexing of 0 to 5 may not be obvious, But look into the
        -- DIGITAL OUT Configuration screen in TP, you will see the
        -- 8 DIGITAL OUTPUT ports are grouped together in
        -- configuration.
        SET_PORT_ASG(IO_DOUT, port_n*8+1, 0, 0, 0, 0, 0, status)
        IF (status <> SUCCESS) AND (status <> UNASIGNED) THEN
            -- Verify that deletion by SET_PORT_ASG was
            -- successful
            WRITE ('SET_PORT_ASG built-in for DOUT (deletion) failed',CR)
            WRITE ('Status = ',status,CR)
        ENDIF
    ENDFOR
    -- Assign the DIGITAL PORTS 1 THRU 48 as memory images.
    FOR port_n = 0 TO 5 DO

```

```

    SET_PORT_ASG(IO_DOUT, port_n*8+1, 0, 0, io_mem_boo, port_n*8+1,
8, status)
    IF (status <> 0 ) THEN      -- Verify that SET_PORT_ASG was
successful
        WRITE ('SET_PORT_ASG built-in for DOUT (assignment)
failed',CR)
        WRITE ('Status = ',status,CR)
    ENDIF
ENDFOR
-- Suppose equipment-1 is turned ON by the DOUT[1] = TRUE signal
-- and turned OFF by the DOUT[2] = TRUE signal. To avoid both
-- signals being TRUE or FALSE at the same time, set DOUT[1] to
-- be a complement. Once the DOUT[1] is set in complementary
-- mode, the DOUT[1] and DOUT[2] will always show the opposite
-- signal of each other. Thus avoiding the confusion of turning
-- the equipment OFF and ON at the same time.
-- Set port-1, port-3 and port-5 to COMPLEMENTARY mode.
FOR port_n = 1 to 6 DO
    SET_MODE(io_dout, port_n, TRUE, TRUE)
    IF (status <> SUCCESS) THEN
        WRITE ('SET_PORT_MODE Failed on port ',1,CR)
        WRITE ('With Status = ',status,CR)
    ENDIF
ENDFOR
-- Set appropriate comments for the ports.
SET_PORT_CMT(IO_DOUT, EQIP_READY, 'Equip-READY ',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, EQIP_NOT_RD, 'E - NOT READY',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, EQIP_ERROR, 'Equip- ERROR',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, EQIP_FIXED, 'Equip- FIXED',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, EQIP_ON, 'Equip- ON',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, EQIP_OFF, 'Equip- OFF',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, NODE_PULSE, 'Pulse @ node',status)
IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
    WRITE ('SET_PORT_CMT built-in failed',CR)
    WRITE ('Status = ',status,CR)
ENDIF
SET_PORT_CMT(IO_DOUT, FINISH, 'Finish PATH',status)

```

```

IF (status <> 0 ) THEN -- Verify SET_PORT_CMT was successful
  WRITE ('SET_PORT_CMT built-in failed',CR)
  WRITE ('Status = ',status,CR)
ENDIF
TP_CLS -- clear the teach pendant USER screen

WRITE ('PORT SETUP IS COMPLETE',CR)
WRITE ('AT THIS POINT YOU NEED TO COLD START',CR)
WRITE ('Configuration changes of PORTS will not',CR)
WRITE ('take effect until after a COLD START.',CR,CR)
WRITE ('Once the controller is ready after',CR)
WRITE ('COLD START, re-load this program',CR)
WRITE ('rerun.',CR)

ports_ready = TRUE      -- Set the ports_ready variable so
                         -- re-execution of this routine,
                         -- setup_ports, is not performed.

-- Aborting program to allow for the cold start.
ABORT
END setup_ports
-----
----- Section 3-E: SERVICE_RTN interrupt routine Declaration
----- This routine waits until the equipment has been
----- serviced and then moves the robot back to where
----- it was before servicing. It then sets the DOUT
----- to notify that the equipment is ready.
-----

ROUTINE service_rtn
BEGIN
  TP_CLS
  -- store the current position, where the process is stopped due to
  -- failure so after resuming the process can be started from
  -- this point.
  stop_pos = CURPOS(0,0)
  -- move the robot to the perch position so the equipment
  -- can be worked on safely.
  SET_POS_REG(1, perch_pos, status) — Put perch_pos in PR[1]
  move_to_pr -- Call TP program to move to PR[1]
  WRITE (chr(139),' PLEASE READ ',chr(143),CR) -- Display in
  WRITE ('Equipment - 1 failed during',CR)      -- reverse video
  WRITE ('processing. Motions have been stopped.',CR)
  WRITE ('Please Fix the equipment then',CR)
  WRITE ('SET DOUT[',EQIP_FIXED,'] = TRUE ',CR)
  --Display the following message in reverse video
  WRITE (chr(139), 'IMPORTANT: Once the DOUT is set, current',CR)
  WRITE ('STOPPED motion will be RESUMED',chr(143),CR)
  WAIT FOR DOUT[EQIP_FIXED] -- wait until equipment has been fixed
  -- Move to the point where the process was stopped
  SET_EPOS_REG(1, stop_pos, status)
  move_to_pr -- Call TP program to move to PR[1]
  -- Enable the SERVICE-DONE condition handler to resume the process.
  ENABLE CONDITION[SERV_DONE]
  -- Wait a sufficient time to allow equipment to warm up and get
  -- ready for processing after the fix is completed.
  WRITE ('Continuing the process.....',CR)
  DELAY 2000

  --Signal that the equipment is now ready.
  DOUT[EQIP_READY] = TRUE

  -- Force the teach pendant back to the IO screen

```

```

    FORCE_SPMENU(tp_panel, SPI_TPDIGIO, 1)
END service_rtn
-----
---- Section 3-F: Routines frst_nod, mid_nods and end_nod are TP
---- routines for doing moves with Time Before clauses
-----
ROUTINE frst_nod FROM frst_nod      -- frst_nod must also be loaded.
-- 1:L PR[1] 100mm/sec CNT100 TB
0.00sec,DO[1:NODE_PULSE]=PULSE,1.0sec ;
-- 2:L PR[1] 100mm/sec CNT100 TB
0.00sec,DO[2:EQUIP_ON]=PULSE,2.0sec ;
ROUTINE mid_nods FROM mid_nods     -- mid_nods must also be loaded.
-- 1:L P[1] 100mm/sec CNT100 TB
0.00sec,DO[1:NODE_PULSE]=PULSE,1.0sec ;
ROUTINE end_nod FROM end_nod       -- end_nod must also be loaded.
-- 1:L P[1] 100mm/sec FINE TB      .20sec,DO[3:FINISH]=ON ;
-- 2:L P[1] 100mm/sec FINE TB
0.00sec,DO[1:NODE_PULSE]=PULSE,1.0sec ;

```

**Figure B.1 (c) Setting Up Digital Output Ports for Process Monitoring - Declare Routines**

```

-----
---- Section 4: Main Program
-----
BEGIN -- DOUT_EX
-----
---- Section 4-A: Global Condition Handler Declaration
-----
CONDITION[UNINIT_CH]:
WHEN ERROR[12311] DO           -- Trap UNINITIALIZATION error
    NOMESSAGE                -- Supress the error message
    UNPAUSE                   -- UNPAUSE
    init_port                 -- Allow ports to be initialized.
ENDCONDITION
ENABLE CONDITION[UNINIT_CH]
CONNECT TIMER to cont_timer
CONDITION [CONT_CH]:
WHEN cont_timer > 1000 DO
    CONTINUE
ENDCONDITION
-----
---- Section 4-B: Verify PATH variable, pth1, has been taught.
-----
tp_cls                         -- Routine Call; Clears the TP USER menu and
                                -- forces the TP USER menu to be visible.
-- Check the number of nodes in the path
IF PATH_LEN(pth1) = 0 THEN      -- Path is empty (no nodes)
    WRITE ('You need to teach the path.',CR) -- Display instructions
    WRITE ('before executing this program.',CR)
    WRITE ('Teach the PATH variable pth1', CR, 'and restart the
program',CR)
    WRITE ('PROGRAM ABORTED',CR)
    ABORT                      -- ABORT the task. do not continue
                                -- There are no nodes to move to
ENDIF
-- Set Perch Position
-- This position is used in the service_rtn routine
IF UNINIT(perch_pos) THEN
    WRITE ('PERCH POSITION is not recorded.',cr)

```

```

        WRITE ('RELEASing Motion Control to TP.',cr)
        WRITE ('Please Move robot to desired Perch Pos',cr)
-- Wait until the DEADMAN switch is HELD and
-- TP is TURNED ON to move robot from TP.
WHILE ((TPIN[248] = ON) AND (TPIN[247] = ON)) DO
    WRITE TPPROMPT(CHR(128),CHR(137),'Hold Down the DEAD-MAN
switch')
    DELAY 500
ENDWHILE
-- Release motion control from the KAREL program to the
-- TP control. Robot can be moved to desired Perch
-- position with out disturbing the flow of this KAREL task.
RELEASE
WHILE (TPIN[249] = OFF ) DO
    WRITE TPPROMPT(CHR(128),CHR(137),'Turn the TP ON')
    DELAY 1000
ENDWHILE
WRITE ('ROBOT is ready to move from TP',cr)
WRITE ('After moving ROBOT to PERCH position ',cr)
WRITE ('Turn OFF the TP then RELEASE DEADMAN ',cr)
WHILE (TPIN[249] = ON ) DO
    WRITE TPPROMPT(CHR(128),CHR(137),'Turn OFF TP, after MOVE is
done
    DELAY 10000
ENDWHILE
-- KAREL program execution will not continue passed ATTACH
-- statement until the TP is turned OFF.
-- Wait until the TP is TURNED OFF after move from TP
-- is completed.
WHILE (TPIN[249] = ON ) DO
    DELAY 2000
ENDWHILE
-- At this point the robot is positioned to the desired
-- Perch position. Get the motion
-- control back from TP and record the perch position.
ATTACH
perch_pos = CURPOS(0,0,1)
ENDIF
-----
----Section 4-C: Set up Ports and Declare Process dependant
----condition handler
-----
-- Port assignments need to be assigned only once and take effect
-- after the controller is COLD STARTED.
-- The ports_ready variable is used to determine if the ports have
-- already been assigned by this program.
-- Therefore only the first execution of this program will assign
-- the ports
IF NOT(ports_ready) THEN
    setup_ports
ENDIF
-- Define a condition handler to trap equipment failure.
-- If equipment fails during the process, then the DOUT[EQIP_ERROR]
-- is set to TRUE. Which will stop the motion and require the
-- equipment to be fixed before motion can be resumed.
CONDITION[EQIP_FAIL]:
WHEN DOUT[EQIP_ERROR] DO
    STOP
    DOUT[EQIP_FIXED] = FALSE
    DOUT[EQIP_READY] = FALSE
    ENABLE CONDITION[RESTART]

```

```

    service_rtn
ENDCONDITION
ENABLE CONDITION[EQIP_FAIL]
-- Define a condition handler to monitor the servicing process.
-- Once Servicing/Fixing of equipment is complete, wait for the
-- equipment to be in READY mode. When the equipment is READY,
-- signal an event which will restart the process where it
-- left off. The SERV_DONE condition handler is ENABLED from
-- the SERVICE_RTN interrupt routine.
CONDITION[SERV_DONE]:
WHEN DOUT[EQIP_READY] DO
    SIGNAL EVENT[WARMED_UP]
    DOUT[EQIP_ERROR] = FALSE
ENDCONDITION

-- Define a condition handler to monitor when the warm up
-- is complete, then resume the stopped motion and continue
-- the process. Also re-enable the EQIP_FAIL condition
-- handler to continue monitoring for equipment failure.
CONDITION[RESTART]:
WHEN EVENT[WARMED_UP] DO
    RESUME
    ENABLE CONDITION[EQIP_FAIL]
ENDCONDITION
-----
----- Section 4-D: Do process manipulation
-----
-- Using the PATH_LEN built-in find out the last node of the path
last_node = PATH_LEN(pth1)
-- Setting EQIP_ERROR/EQIP_FIXED number ports to be simulated.
-- This setup does not require cold start, can change the port to be
-- simulated on the fly.
SET_PORT_SIM(io_dout, NODE_PULSE, 1, status)
IF (status <> SUCCESS) THEN
    WRITE ('SET_PORT_SIM Failed on port ',idx,CR)
    WRITE ('With Status = ',status,CR)
ENDIF
SET_PORT_SIM(io_dout, FINISH, 1, status)
IF (status <> SUCCESS) THEN
    WRITE ('SET_PORT_SIM Failed on port ',idx,CR)
    WRITE ('With Status = ',status,CR)
ENDIF
WRITE (' NOW YOU WILL SEE THE DOUT[,NODE_PULSE,'] PULSE',CR)
WRITE (' as the robot moves through every node.',CR,CR)
WRITE (' To simulate EQUIPMENT failure, change ',CR)
WRITE (' DOUT[,EQIP_ERROR,'] = TRUE. ',CR)
WRITE (' Press ''ENTER'' to Continue',CR)
READ(CR)
-- Change the TP display to the DI/O Screen
FORCE_SPMENU(tp_panel, SPI_TPDIGIO, 1)
-- Moving along path when equipment is ready.
-- Need to turn on equipment-1 for 1/2 second when robot position
-- is at 1st node. Pulse the DOUT[NODE_PULSE] for every node
-- Turn on the DOUT[FINISH] about 200 ms before the last node.
IF DOUT[EQIP_READY] THEN
    tmp_xyz = pth1[1] -- Convert path node to XYZWPR
    SET_POS_REG(1, tmp_xyz, status) -- Put position in PR[1]
    frst_nod -- Call TP program to do move
    FOR node_ind = 2 TO (last_node - 1) DO
        tmp_xyz = pth1[node_ind]
        SET_POS_REG(1, tmp_xyz, status)

```

```

    mid_nods
ENDFOR
tmp_xyz = pth1[last_node]
SET_POS_REG(1, tmp_xyz, status)
end_nod
ELSE
FORCE_SPMENU(TP_PANEL,SPI_TPUSER,1)
WRITE (' Equipment is not READY',CR)
WRITE (' Set equipment to READY MODE',CR)
WRITE (' before executing this program.',CR)
WRITE (' SET DOUT[,EQIP_READY,'] = TRUE ',CR)
ABORT
ENDIF
WRITE TPFUNC      (CHR(128),CHR(137)) -- Home Cursor and Clear to
                                         -- End-of-line This will
                                         -- remove the ABORT displayed
                                         -- above F1.
END DOUT_EX

```

**Figure B.1 (d) Setting Up Digital Output Ports for Process Monitoring - Main**

## B.2 COPYING PATH VARIABLES

This example shows the different ways of copying and appending PATH variables. The PATH data type can be copied from one to another only with hard coded path variable names. However, user defined paths can be copied from one to another. The path variable names can be determined during execution of the program.

```

-----
---- Detail about CPY_PTH.K1
-----
---- Elements of KAREL Language Covered: In Section:
---- Action:
---- Clauses:
----          FROM                      Sec 3-A
----          IN DRAM                  Sec 2
----          WHEN                     Sec 4-A
---- Conditions:
---- Data types:
----          ARRAY OF STRING        Sec 2
----          BOOLEAN                 Sec 2; 3-C
----          FILE                    Sec 2
----          INTEGER                 Sec 2; 3-B,C
----          PATH                    Sec 2
----          STRING                  Sec 2; 3-B
----          STRUCTURE...ENDSTRUCTURE Sec 2
----          USER DEFINED PATH       Sec 2
----          XYZWPR                 Sec 2
---- Directives:
----          ALPHABETIZE            Sec 1
----          COMMENT                Sec 1
----          CMOSVARS               Sec 1
----          CRTDEVICE              Sec 1
----          INCLUDE                Sec 2
---- Built-in Functions & Procedures:
----          APPEND_NODE            Sec 4-D
----          BYNAME                 Sec 4-E
----          CALL_PROG              Sec 4-B

```

-----	COPY_PATH	Sec 3-C; 4-D
-----	CNV_INT_STR	Sec 4-E
-----	CREATE_VAR	Sec 4-E
-----	CURPOS	Sec 4-B
-----	DELETE_NODE	Sec 4-C
-----	LOAD	Sec 4-B
-----	PATH_LEN	Sec 4-C, E
-----	PROG_LIST	Sec 4-B
-----	READ_KB	Sec 3-B
-----	SET_CURSOR	Sec 4-E
-----	SET_FILE_ATR	Sec 4-A
-----	SET_POS_REG	Sec 4-D
-----	SET_VAR	Sec 4-B
-----	SUB_STR	Sec 4-E
-----	VAR_LIST	Sec 4-E
-----	Statements:	
-----	ABORT	Sec 4-C, E
-----	CLOSE FILE	Sec 4-E
-----	FOR .... ENDFOR	Sec 3-C; 4-C, D, E
-----	IF...THEN...ENDIF	Sec 3-B, C; 4-B, C, D, E
-----	OPEN FILE	Sec 4-A
-----	REPEAT...UNTIL	Sec 3-B; 4-E
-----	ROUTINE	Sec 3
-----	WRITE	Sec 3-B, C; 4-A, B, C, E
-----	USING...ENDUSING	Sec 4-D
-----	Reserve Word:	
-----	BEGIN	Sec 3-B, C; 4
-----	END	Sec 3-B, C; 4-E
-----	PROGRAM	Sec 1
-----	TYPE	Sec 2
-----	VAR	Sec 2
-----	Predefined File Names:	
-----	CRTFUNC	Sec 3-B
-----	CRTPROMPT	Sec 3-B, C

**Figure B.2 (a) Copy Path Variables Program - Overview**

```

----- Section 1: Program and Environment Declaration
-----
PROGRAM CPY_PTH
%ALPHABETIZE
%COMMENT = 'COPY PATH'           -- Display information by default to
                                  -- CRT/KB
%CRTDEVICE
%CMOSVARS                      -- Use CMOS RAM to store all static
                                  -- variables, except those specified
                                  -- with IN DRAM
-----
----- Section 2: Constant, Variable and Type Declarations
-----
CONST
  SUCCESS      = 0 -- The value returned from all built-ins
                    -- when successful
TYPE
  node_struc   = STRUCTURE    -- Create a user defined node
    posn_dat    :XYZWPR      -- structure
  ENDSTRUCTURE
  user_path     = PATH nodedata = node_struc --Create a user

```

```

--defined path
VAR
  pth1,
  pth2,
  pth3
  pth4          :PATH      -- These are system defined PATHs
  upth1,
  upth2,
  upth3,
  upth4          :user_path -- These are user defined PATHs

  p1_len,
  p2_len,
  status, node_ind,
  total_node      :INTEGER
  F1_press,
  F2_press        :BOOLEAN
  src_num,
  des_num         :INTEGER
  dummy_str,
  src_var,
  des_var         :STRING[20]
  cur_name        :STRING[12]
  entry           :INTEGER
  var_type        :INTEGER
  mem_loc         :INTEGER

```

**Figure B.2 (b) Copy Path Variables Program - Declaration Section**

```

-- Store the following variables in DRAM, which is temporary memory
  indx      IN DRAM   :INTEGER
  prog_name IN DRAM   :STRING[10]
  prog_type IN DRAM   :INTEGER
  n_match   IN DRAM   :INTEGER
  n_skip    IN DRAM   :INTEGER
  format    IN DRAM   :INTEGER
  ary_nam   IN DRAM   :ARRAY[5] OF STRING[20]
  prog_indx IN DRAM   :INTEGER
  do_copy   IN DRAM   :BOOLEAN
  crt_kb    IN DRAM   :FILE
%INCLUDE KLEVMSK      -- system supplied file: definition of
                      -- KC_FUNC_KEY
%INCLUDE KLEVKEYS     -- system supplied file: definition of
                      -- KY_F1 & KY_F2

```

**Figure B.2 (c) Copy Path Variables Program - Storing Variables in Memory**

```

-----
----- Section 3: Routine Declaration
-----

-----
----- Section 3-A: CRT_CLS Declaration
-----

ROUTINE CRT_CLS FROM rout_ex -- include this routine from the file
                           -- rout_ex.kl
-----

----- Section 3-B: YES_NO Declaration
----- LABEL the F1 key as YES and F2 key as NO, ask
----- for user confirmation. These two keys are

```

```

---- monitored by global condition handler, so User
---- response can be trapped.
-----
ROUTINE YES_NO
VAR
    key_press : INTEGER
    str : STRING[1]
    n_chars: INTEGER
    l_status: INTEGER
BEGIN ---- YES_NO
    WRITE CRTFUNC    (CHR(128),CHR(137)) --- Clear Window, Home Cursor
    -- Display YES above F1 & NO above F2 & clear rest of Function
    -- window
    WRITE CRTFUNC    ('    YES      NO ',chr(129))
    F1_press = FALSE
    F2_press = FALSE
    REPEAT -- until user presses either the F1 or F2 key
        -- Read just the function keys of the CRT/KB.
        -- The read will be satisfied only when a function key is
        -- pressed.
        READ_KB (crt_kb, str , 0, 0, kc_func_key, -1,
                  ', n_chars, key_press, l_status)
        -- key_press must be converted from a "raw" CRT character to the
        -- teach pendant equivalent character.g
        key_press = $CRT_KEY_TBL[key_press+1]
        IF (key_press = ky_f1) THEN -- The user pressed F1
            F1_press = true
        ENDIF
        IF (key_press = ky_f2) THEN -- The user pressed F2
            F2_press = true
        ENDIF
    UNTIL ((f1_press = TRUE) OR (F2_press = TRUE))
    WRITE CRTFUNC    (CHR(128),CHR(137)) --- Clear Window, Home Cursor
    WRITE CRTPROMPT  (CHR(128),CHR(137)) --- Clear Window, Home Cursor
END YES_NO

```

**Figure B.2 (d) Copy Path Variables Program - Monitor User Response**

```

----- Section 3-C: PTH_CPY Declaration
----- Copy one user defined path variable to another user
----- defined path variable. The first parameter is the
----- source path. The second parameter is the destination
----- path. The path parameters can only be passed using
----- BYNAME and the paths must be user defined
-----
ROUTINE PTH_CPY(src_path: USER_PATH; des_path: USER_PATH)
VAR
    node_indx  :INTEGER
    do_it       :BOOLEAN
    l_stat      :INTEGER
BEGIN --- pth_cpy
    CRT_CLS      -- Clear the CRT/KB USER Menu screen
    do_it = true
    WRITE ('Perform copy?',CR)
    yes_no
    do_it = F1_press -- F1_press will be true only if the user
                      -- selected
YES

```

```

IF (do_it) THEN
    -- Copy the entire path of src_path to des_path
    COPY_PATH (src_path, 0,0, des_path, l_stat)
    IF (l_stat <> 0) THEN
        WRITE ('Error in COPY_PATH', l_stat, CR)
    ELSE
        WRITE ('Path Copy function Completed ',cr)
    ENDIF
ELSE
    WRITE ('Path Copy function canceled by choice',cr,cr)
ENDIF
END PTH_CPY
-----
---- Section 3-D: Routine move_to_pr is a TP
---- routine for doing moves
-----
ROUTINE move_to_pr FROM move_to_pr      -- move_to_pr must also be
                                         -- loaded.
-- 1:J PR[1] 100% FINE      ;

```

**Figure B.2 (e) Copy Path Variables Program - Copying Path Variables**

```

-----
---- Section 4: Main Program
-----
BEGIN --- CPY_PATH
-----
---- Section 4-A: Open CRT KB for reading YES/NO inputs from user
-----
CRT_CLS -- will force the CRT USER menu to be visible & clear the
          -- screen
SET_FILE_ATR(crt_kb, ATR_FIELD) -- Needed so the read is satisfied
                                  -- with one character.
OPEN FILE crt_kb ('RO', 'KB:crkb') -- Open a file to the CRT/KB
                                     -- Used within the YES_NO
                                     -- routine.
-----
---- Section 4-B: Check if SAVE_VRS.PC is loaded. If loaded then
---- execute
-----
---- First check if the "SAVE_VRS" program is loaded or not.
prog_name = 'SAVE_VRS' -- Only interested in SAVE_VRS program
prog_type = 6           -- Interested only in PC type files
n_skip = 0              -- First time do not skip any files
format = 1              -- Return just the filename
do_copy = TRUE
WRITE ('Checking Program List',cr)
PROG_LIST(prog_name, prog_type, n_skip, format, ary_nam, n_match,
status)
IF (status <> SUCCESS ) THEN
    IF (status = 7073 ) THEN ---- Program does not exist error
        -- Program SAVE_VRS is not loaded on the controller.
        WRITE ( 'LOADING ',prog_name, CR)
        LOAD (prog_name+'.PC', 0, status)
        IF (status <> SUCCESS) THEN
            WRITE ('Error loading ', prog_name,cr)
            WRITE CRTPROMPT('Copy paths WITHOUT saving program
variables?',CR)
            YES_NO

```

```

        do_copy = F1_press -- F1_press is true only if user selected
        -- YES
        -- Copy without saving variables.
    ENDIF
ELSE
    -- The program listing failed.
    WRITE ('PROG_LIST built-in failed',cr,' with Status =
',status,cr)
    WRITE CRTPROMPT('Copy paths WITHOUT saving program
variables?',CR)
    YES_NO
    do_copy = F1_press -- F1_press is true only if user selected
    -- YES
ENDIF
    -- Copy without saving variables.
ENDIF

```

**Figure B.2 (f) Copy Path Variables Program - Opens CRT/KB & Sends Data to Default Device**

```

IF (status = SUCCESS) THEN
    -- This is one way to set variables within another program without
    -- using the FROM clause in the variable section.
    -- It is very useful if you want to have run-time independent code,
    -- where the program or variable name you are setting is not
    -- known until run-time.
    cur_name = CURR_PROG
    SET_VAR (entry, prog_name, 'del_vr', TRUE, status)
    SET_VAR (entry, prog_name, 'prog_name', cur_name , status)
    SET_VAR (entry, prog_name, 'sav_type', 1, status)
    SET_VAR (entry, prog_name, 'dev', 'FLPY:', status)
    WRITE ('Saving program variables before copy', CR)
    CALL_PROG(prog_name,prog_indx) -- call SAVE_VRS
ENDIF
-----
----- Section 4-C: Check for initialization of PATHs pth1 and pth2.
-----
IF (NOT do_copy) THEN
    WRITE ('Program exiting, unable to save variables,',cr)
    WRITE ('before copying path''s content',cr)
    -- NOTICE:
    -- Two single quotes will display as one single quote
    -- so this write statement will appear as :
    -- "before copying path's content"
    ABORT
ENDIF
WRITE ('Checking Variable initialization',cr)
-- Check if the pth variables are initialized.
pth1_len = PATH_LEN(pth1) ; pth2_len = PATH_LEN(pth2)
IF ( (pth1_len = 0) OR (pth2_len = 0) )THEN
    WRITE ('PTH1 or PTH2 is empty path',cr)
    WRITE ('Please make sure both paths are taught then restart',cr)
    ABORT -- Cannot copy uninitialized variables.
ENDIF
-- Check if the pth3 variable is initialized.
IF (PATH_LEN(pth3) <> 0) THEN
    WRITE ('Deleting nodes from pth3',cr) -- Delete the old path
    -- of pth3
    FOR indx = PATH_LEN(pth3) DOWNTO 1 DO
        -- its easy to delete nodes from the end instead of deleting
        -- node from the front end. Since after every deletion the
        -- nodes are renumbered.

```

```

DELETE_NODE(pth3, indx, status) -- Delete last node of pth3
IF status <> SUCCESS THEN
    WRITE ('While Deleting ',indx, ' node',cr)
    WRITE ('DELETE_NODE unsuccessful: Status = ',status,cr)
ENDIF
ENDFOR
ENDIF

```

**Figure B.2 (g) Copy Path Variables Program - Checks Path Initialization**

```

-----  

---- Section 4-D: Add pth1 and pth2 together to create pth3.  

---- Move along pth1 and pth2.  

---- Move backwards through pth3.  

-----  

total_node = p1_len + p2_len      -- Total number of nodes needed for  

                                    -- pth3  

-- Copy the node data from pth1 to pth3  

WRITE ('copying pth1 to pth3',cr)  

COPY_PATH (pth1, 0,0, pth3, status)  

    IF (status <> 0) THEN  

        WRITE ('ERROR in COPY_PATH', status, CR)  

    ENDIF  

-- Create the required number of nodes for pth3.  

-- We know that pth3 now has PATH_LEN(pth3) nodes.  

WRITE ('Appending nodes to pth3',cr)  

FOR indx = p1_len+1 TO total_node DO -- Append the correct number  

    APPEND_NODE(pth3, status)           -- of nodes.  

    IF (status <> 0) THEN  

        WRITE ('While Appending ',indx, ' node',cr)
        WRITE ('APPEND_NODE unsuccessful: Status = ',status,cr)
    ENDIF  

ENDFOR  

-- Append the node data of pth2 to pth3.  

WRITE ('Appending pth2 to pth3',cr)  

FOR indx = p1_len+1 TO total_node DO  

    USING pth2[indx - p1_len] DO  

        pth3[indx].node_pos = node_pos  

    ENDUSING  

ENDFOR  

-- Move along the path pth1 and pth2  

WRITE ('Moving Along Path pth1',cr)  

FOR node_ind = 1 TO p1_len DO  

    tmp_xyz = pth1[node_ind]  

    SET_POS_REG(1, tmp_xyz, status)  

    move_to_pr -- Call TP program to move to PR[1]
ENDFOR  

WRITE ('Moving Along Path pth2',cr)  

FOR node_ind = 1 TO p2_len DO  

    tmp_xyz = pth2[node_ind]  

    SET_POS_REG(1, tmp_xyz, status)  

    move_to_pr -- Call TP program to move to PR[1]
ENDFOR  

--Copy pth3 in reverse order to pth4  

COPY_PATH (pth3, PATH_LEN(pth3), 1, pth4, status)  

IF (status <> 0) THEN  

    WRITE ('ERROR in COPY_PATH', status, CR)
ENDIF--- Move along pth4 which is a reverse order of pth3.  

WRITE ('Moving Along Path pth4',cr)

```

```

FOR node_ind = 1 TO PATH_LEN(pth4) DO
    tmp_xyz = pth4[node_ind]
    SET_POS_REG(1, tmp_xyz, status)
    move_to_pr -- Call TP program to move to PR[1]
ENDFOR

```

**Figure B.2 (h) Copy Path Variables Program - Path Initialization**

```

-----  

---- Section 4-E: Copy User Defined Paths.  

---- Copy one user defined path to another user  

---- defined path, where the user specifies which  

---- paths to be copied.  

----  

CRT_CLS  

SET_CURSOR(OUTPUT,2,10, status) -- Position cursor nicely on CRT  

IF (status <> 0 ) THEN  

    WRITE ('SET_CURSOR built-in failed with status = ',status,cr)  

ENDIF  

-- write message in reverse video and then set back to normal video  

WRITE (chr(139),' COPY PATH FUNCTION',chr(143),CR,cr)  

WRITE ('Currently you have the following ',cr)  

WRITE ('User Defined Paths',cr,cr)  

n_skip = 0  

var_type = 31 -- Get listing of only PATH type variables  

REPEAT  

VAR_LIST ('CPY_PTH', '*',var_type, n_skip, 2, ary_nam, n_match,  

status)  

    FOR indx = 1 TO n_match DO  

        IF (SUB_STR (ary_nam[indx], 1, 4) = 'UPTH') THEN -- Verify it's  

            -- one of the user  

            -- defined paths  

            WRITE (ary_nam[indx], CR)  

        ENDIF  

    ENDFOR  

    n_skip = n_skip + n_match  

UNTIL (n_match < ARRAY_LEN(ary_nam))  

Write ('Enter the source path number:')  

READ(src_num);  

Write ('Enter the destination path number:')  

READ(des_num);  

CNV_INT_STR(src_num,2,0,dummy_str) -- Convert source number  

-- to string  

src_var = 'UPTH'+ SUB_STR(dummy_str,2,1) -- SUB_STR will remove the  

-- leading blank from  

-- dummy_str before  

-- concatenating to create  

-- the source variable name  

var_type = 0  

VAR_LIST ('CPY_PTH', src_var, var_type, 0, 2, ary_nam, n_match,  

status)  

IF (status <> SUCCESS) THEN  

    WRITE ('Var_list unsuccessful for src_var: status ', status, cr)  

ENDIF  

-- If the variable does not exist create it.  

IF (n_match = 0) THEN  

    CREATE_VAR ('', src_var, '', 'USER_PATH', 1, 0, 0, 0, status,  

mem_loc)  

    IF (status <> SUCCESS) THEN  

        WRITE ('Error creating ', src_var, ':', status, cr)  

    ENDIF

```

```

ENDIF
--Create the destination variable name
CNV_INT_STR(des_num,2,0,dummy_str)           -- Convert des_num to a
                                                -- string
des_var = 'UPTH'+ SUB_STR(dummy_str,2,1) -- The SUB_STR will remove
                                            -- the leading blank from
                                            -- dummy_str before
                                            -- concatenating to create
                                            -- the source variable name
-- Verify that the des_var variable exists.
VAR_LIST ('CPY_PTH', des_var, var_type, 0, 2, ary_nam, n_match,
status)
IF (status <> SUCCESS) THEN
    WRITE ('Var_list unsuccessful for des_var: status', status, cr)
ENDIF
-- If the variable does not exist create it.
IF (n_match = 0) THEN
    CREATE_VAR ('', des_var, '', 'USER_PATH', 1, 0, 0, 0, status,
mem_loc)
    IF (status <> SUCCESS) THEN
        WRITE ('Error creating ', des_var, ':', status, cr)
    ENDIF
ENDIF
-- Copy the specified source path to the specified destination path
pth_cpy(BYNAME('', src_var, indx), BYNAME('', des_var, indx) )
-- Close file before quitting
CLOSE FILE crt_kb
WRITE ('CPY_PTH example completed',cr)
END CPY_PTH

```

**Figure B.2 (i) Copy Path Variables Program - Copy User Defined Paths**

## B.3 SAVING DATA TO THE DEFAULT DEVICE

This program will save variables or teach pendant programs to the default device. If the user specified to overwrite the file then the file will be deleted before performing the save.

```

-----
---- SAVE_VRS.KL
-----
---- Section 0: Detail about SAVE_VRS.KL
-----
---- Elements of KAREL Language Covered: In Section:
---- Actions:
---- Clauses:
---- Conditions:
---- Data types:
----          BOOLEAN             Sec 2
----          INTEGER            Sec 2
----          STRING             Sec 2
---- Directives:
----          COMMENT            Sec 1
----          ENVIRONMENT        Sec 1
----          NOLOCKGROUP        Sec 1
---- Built-in Functions & Procedures:
----          DELETE_FILE         Sec 4-B
----          SAVE               Sec 4-B
-----
```

```
---- Statements:
----      IF, THEN, ENDIF          Sec 4-B
----      SELECT, CASE, ENDSELECT   Sec 4-A
----      WRITE                   Sec 4-B
---- Reserve Words:
----      BEGIN                  Sec 4
----      CONST                  Sec 2
----      CR                     Sec 4-B
----      END                     Sec 4-B
----      PROGRAM                Sec 1
----      VAR                     Sec 2
```

**Figure B.3 (a) Saving Data Program - Overview**

```
-----  
---- Section 1: Program and Environment Declaration  
-----  
PROGRAM SAVE_VRS  
%NOLOCKGROUP  
%COMMENT = 'Save .vr, .tp, .sv'  
%ENVIRONMENT MEMO  
%ENVIRONMENT FDEV  
-----  
---- Section 2: Constant, Variable and Type Declarations  
-----  
CONST  
  DO_VR  = 1      -- Save variable file(s)  
  DO_TP  = 2      -- Save TP program(s)  
  DO_SYS = 3      -- Save system variables  
  SUCCESS = 0     -- The value expected from all built-in calls.  
VAR  
  sav_type : INTEGER    -- Specifies the type of save to perform  
  prog_name : STRING[12] -- The program name to save  
  status     : INTEGER    -- The status returned from the built-in  
  calls  
  file_spec : STRING[30]  -- The created file specification for SAVE  
  dev       : STRING[5]   -- The device to save to specify whether to  
  del_vr    : BOOLEAN     -- delete file_spec before performing the  
                         -- SAVE.  
-----  
---- Section 3: Routine Declaration  
-----
```

**Figure B.3 (b) Saving Data Program - Declarations Section**

```
-----  
---- Section 4: Main Program  
-----  
BEGIN -- SAVE_VRS  
-----  
---- Section 4-A: Create the file_spec, which contains the  
----               device, file name and type to be saved.  
-----  
  SELECT (sav_type) OF  
  CASE (DO_VR):  
    -- If prog_name is '*' then all PC variables will be saved with  
    -- the correct program name, regardless of the file name part  
    -- of file_spec.  
    file_spec = dev+prog_name+'.VR' -- Create the variable file name
```

```

CASE (DO_TP):
-- If prog_name is '*' then all TP programs will be saved with
-- the correct TP program name, irregardless of the prog_name
-- part of file_spec.
file_spec = dev+prog_name+'.TP' -- Create the TP program name
CASE (DO_SYS):
prog_name = '*SYSTEM*'
file_spec = dev+'ALLSYS.SV' -- All system variables will be
-- saved into this one file.
ENDSELECT

```

**Figure B.3 (c) Saving Data Program - Create File Spec**

```

-----  

---- Section 4-B: Decide whether to delete the file before saving  

---- and then perform the SAVE.  

-----  

-- If the user specified to delete the file before saving, then  

-- delete the file and verify that the delete was successful.  

-- It is possible that the delete will return a status of:  

-- 10003 : "file does not exist", for the FLPY: device  

-- OR  

-- 85014 : "file not found", for all RD: and FR: devices  

-- We will disregard these errors since we do not care if the  

-- file did not previously exist.  

IF (del_vr = TRUE) THEN  

    DELETE_FILE (file_spec, FALSE, status) -- Delete the file.  

    IF (status <> SUCCESS) AND (status <> 10003) AND  

        (status <> 85014) THEN  

        WRITE ('Error ', status, ' in attempt to delete ', cr,  

file_spec, cr)
    ENDIF  

ENDIF  

-- If prog_name is specified as an '*' for either .tp or .vr  

-- files then the SAVE built-in will save the appropriate  

-- files/programs with the correct names.  

SAVE (prog_name, file_spec, status) -- Save the variable/program  

IF (status <> SUCCESS) THEN -- Verify SAVE was successful  

    WRITE ('error saving ', file_spec, 'variables', status, cr)
ENDIF  

END SAVE_VRS

```

**Figure B.3 (d) Saving Data Program - Delete/Overwrite**

## B.4 STANDARD ROUTINES

---

This program is made up of several routines which are used through out the examples. The following is a list of the routines within this file:

- **CRT\_CLS** – Clears the CRT/KB **USER Menu** screen
- **TP\_CLS** – Clears the teach pendant **USER Menu** screen

```

-----  

---- ROUT_EX.KL  

-----  

---- Section 0: Detail about ROUT_EX.kl  

-----
```

---- Elements of KAREL Language Covered:	In Section:
---- Actions:	
---- Clauses:	
---- Conditions:	
---- Data types:	
---- Built-in Functions & Procedures:	
----      CHR	Sec 3-A,B
----      FORCE_SPMENU	Sec 3-A,B
---- Statements:	
----      ROUTINE	Sec 3-A,B
----      WRITE	Sec 3-A,B
---- Reserve Words:	
----      BEGIN	Sec 3-A,B; 4
----      CR	Sec 3-B
----      END	Sec 3-A,B; 4
----      PROGRAM	Sec 1
---- Predefined File Names:	
----      CRTERror	Sec 3-A
----      CRTFUNC	Sec 3-A
----      CRTPROMPT	Sec 3-A
----      CRTSTATUS	Sec 3-A
----      OUTPUT	Sec 3-A
----      TPERROR	Sec 3-B
----      TPFUNC	Sec 3-B
----      TPSTATUS	Sec 3-B
----      TPPROMPT	Sec 3-B

**Figure B.4 (a) Standard Routines - Overview**

```

-----  

---- Section 1: Program and Environment Declaration  

-----  

PROGRAM ROUT_EX  

%NOLOCKGROUP ----- Don't lock any motion groups  

%COMMENT = 'MISC_ROUTINES'  

-----  

---- Section 2: Constant and Variable Declarations  

-----  

-----  

---- Section 3: Routine Declarations  

-----  

-----  

---- Section 3-A: CRT_CLS Declaration  

----      Clear the predefined windows:  

----      CRTPROMPT, CRTSTATUS, CRTFUNC, CRTERror, OUTPUT  

----      Force Display of the CRT/KB USER SCREEN.  

-----  

ROUTINE CRT_CLS  

BEGIN ---- CRT_CLS  

--See Chapter 7.9.2 for more information on the PREDEFINED window  

--names  

  WRITE CRTERror    (CHR(128),CHR(137))    -- Clear Window, Home Cursor  

  WRITE CRTSTATUS   (CHR(128),CHR(137))    -- Clear Window, Home Cursor  

  WRITE CRTPROMPT   (CHR(128),CHR(137))    -- Clear Window, Home Cursor  

  WRITE CRTFUNC    (CHR(128),CHR(137))    -- Clear Window, Home Cursor  

  WRITE OUTPUT    (CHR(128),CHR(137))    -- Clear Window, Home Cursor  

  FORCE_SPMENU(CRT_PANEL,SPI_TPUSER,1)    -- Force the CRT USER Menu  

                                                  -- to be visible last. This  

                                                  -- will avoid the screen from

```

```

-- flashing since the screen
-- will be clean when you
-- see it.

END CRT_CLS

```

**Figure B.4 (b) Standard Routines - Declaration Section**

```

-----  

---- Section 3-B:    TP_CLS Declaration  

                    Clear the predefined windows:  

----                      TPERROR, TPSTATUS, TPPROMPT, TPFUNC TPDISPLAY  

----                      Force Display of the TP USER Menu SCREEN.  

-----  

ROUTINE TP_CLS  

BEGIN  

    WRITE (CHR(128),CHR(137)) -- By default this will clear TPDISPLAY  

    WRITE TPERROR (CR,'           ',CR)  

    WRITE TPSTATUS (CR,'           ',CR)  

    WRITE TPPROMPT(CR,'           ',CR)  

    WRITE TPFUNC   (CR,'           ',CR)  

    FORCE_SPMENU(TP_PANEL,SPI_TPUSER,1) -- Force the USER menu screen  

        -- to be visible last.  

        -- This will avoid the screen from  

        -- flashing since the screen will  

        -- be clean when you see it.  

END TP_CLS  

-----  

---- Section 4: Main Program  

-----  

BEGIN -- ROUT_EX  

END ROUT_EX

```

**Figure B.4 (c) Standard Routines - Clears Screen and Displays Menu**

## B.5 USING REGISTER BUILT-INS

This program demonstrates the use of the REGISTER built-ins. REG\_EX.KL retrieves the current position and stores it in PR[1]. Then it executes the program PROG\_VAL.TP. PROG\_VAL will modify the value within the Position Register PR[1].

After PROG\_VAL is completed, REG\_EX.KL retrieves the PR[1] position. The position is then manipulated and restored in PR[2], and an INTEGER number is stored in R[1]. A different teach pendant program, PROG\_1.TP, is executed which loops through some positions and stores a value to R[2]. The number of loops depends on the value of the R[1] (which was initially set by the KAREL program.)

After PROG\_1.TP has completed, the KAREL program gets the value from R[2] and verifies it was the expected value.

The PROG\_VAL.TP teach pendant program should look similar to the following.

PROG_VAL	JOINT 10%
<pre> 1: !POSITION REG VALUE ; 2:J P[1:ABOVE JOINT] 100% FINE ; 3: PR[1,2]=600 ; 4:L PR[1] 100.0 Inch/mm FINE ; 5:J P[1:ABOVE JOINT]100% FINE ; </pre>	

The PROG\_VAL.TP teach pendant program does the following:

- Moves to position 1 in joint mode.
- Changes the y location of the position in Position Register 1, PR[1] (which was set by the KAREL program).
- Moves to the new PR[1] position.
- Finally moves back to position 1.

The PROG\_1.TP teach pendant program should look similar to the following.

PROG_1	JOINT 10%
<pre> 1: LBL[1:START] ; 2: IF R[1]=0, JMP LBL[2] ; 3:J P[1] 100% FINE ; 4:J P[2] 100% FINE ; 5: R[1]=R[1]-1 ; 6: JMP LBL[1] ; 7: LBL[2:DONE] ; 8: R[2]=1 ; </pre>	

The PROG\_1.TP teach pendant program does the following:

- Checks the value of the R[1].
- If the value of R[1] is not 0, then moves to J P[1] and J P[2] and decrements the value of R[1]. PROG\_1.TP continues in this loop until the Register R[1] is zero.
- After the looping is complete, PROG\_1.TP stores value 1 in R[2], which will be checked by the KAREL program.

<hr/>	
---- REG_EX.K1	
<hr/>	
---- Elements of KAREL Language Covered:	In Section:
---- Actions:	
---- Clauses:	
---- Conditions:	
---- Data types:	
----           BOOLEAN	Sec 2
----           JOINTPOS	Sec 2
----           REAL	Sec 2
----           XYZWPR	Sec 2
---- Directives:	
----           ALPHABETIZE	Sec 1
----           COMMENT	Sec 1
----           NOLOCKGROUP	Sec 1
---- Built-in Functions & Procedures:	

-----	CALL_PROGLIN	Sec 4-A, 4-C
-----	CHR	Sec 4
-----	CURPOS	Sec 4-A
-----	FORCE_SPMENU	Sec 4
-----	GET_POS_REG	Sec 4-B
-----	GET_JPOS_REG	Sec 4-B
-----	GET_REG	Sec 4-C
-----	POS_REG_TYP	Sec 4-B
-----	SET_JPOS_REG	Sec 4-B
-----	SET_INT_REG	Sec 4-B
-----	SET_POS_REG	Sec 4-A
 Statements:		
-----	WRITE	Sec 4, 4-A,B,C
-----	IF..THEN..ELSE..ENDIF	Sec 4-A,B,C
-----	SELECT...CASE...ENDSELECT	Sec 4-B
 Reserve Words:		
-----	BEGIN	Sec 4
-----	CONST	Sec 2
-----	CR	Sec 4-A,B,C
-----	END	Sec 4-C
-----	PROGRAM	Sec 4
-----	VAR	Sec 2

**Figure B.5 (a) Using Register Built-ins Program - Overview**

```

----- Section 1: Program and Environment Declaration
-----
PROGRAM reg_ex
%nolockgroup
%comment = 'Reg-Ops'
%alphabetize
-----
----- Section 2: Variable Declaration
-----
CONST
  cc_success      = 0      -- Success status
  cc_xyzwpr       = 2      -- Position Register has an XYZWPR
  cc_jntpos       = 9      -- Position Register has a JOINTPOS
VAR
  xyz             :XYZWPR
  jpos            :JOINTPOS
  r_val           :REAL
  prg_indx,
  i_val,
  pos_type,
  num_axes,
  status          :INTEGER
  r_flg           :BOOLEAN
-----
----- Section 3: Routine Declaration
-----
-----
----- Section 4: Main program
-----
BEGIN -- REG_EX
  write(chr(137),chr(128));           -- Clear the TP USER menu screen

```

```
FORCE_SPMENU(TP_PANEL,SPI_TPUSER,1) -- Force the TP USER menu to be
-- visible
```

**Figure B.5 (b) Using Register Built-ins Program - Declaration Section**

```
-----
---- Section 4-A: Store current position in PR[1] and execute
PROG_VAL.TP
-----

WRITE('Getting Current Position',cr)
xyz = CURPOS(0,0) -- Get the current position
WRITE('Storing Current position to PR[1]',cr)
SET_POS_REG(1,xyz, status) -- Store the position in PR[1]
IF (status = cc_success) THEN -- verify SET_POS_REG is
-- successful
WRITE('Executing "PROG_VAL.TP"',cr)
CALL_PROGLIN('PROG_VAL',2,prg_idx, FALSE)
-- Execute 'PROG_VAL.TP'
-- starting at line 2.
-- Do not pause on
-- entry of PROG_VAL.

-----
---- Section 4-B: Get new position from PR[1]. Manipulate and store
---- in PR[2]
-----

WRITE('Getting Position back from PR[1]',cr)
-- Decide what type of position is stored in Position
-- Register 1, PR[1]
POS_REG_TYPE(1, 1, pos_type, num_axes, status)
IF (status = cc_success) THEN
-- Get the position back from PR[1], using the correct built-in.
-- This position was modified in PROG_VAL.TP
SELECT pos_type OF
CASE (cc_xyzwpr):
    xyz= GET_POS_REG(1, status)
CASE (cc_jntpos):
    jpos = GET_JPOS_REG(1, status)
    xyz = jpos
ELSE:
    write ('The position register set to invalid type',
pos_type,CR)
    status = -1 -- set status so do not continue.
ENDSELECT
IF (status = cc_success) THEN -- Verify GET_POS_REG/
-- GET_JPOS_REG is successful
    xyz.x = xyz.x+10 -- Manipulate the position.
    xyz.z = xyz.z-10
    jpos = xyz -- Convert to a JOINTPOS
    WRITE('Setting New Position to PR[2]',cr)
    SET_JPOS_REG(2,jpos,status) -- Set the JOINTPOS into PR[2]
    IF (status = cc_success) THEN -- Verify SET_JPOS_REG is
-- successful
        WRITE('Setting Integer Value to R[1]',cr)
        SET_INT_REG(1, 10, status) -- Set the value 10 into R[1]
```

**Figure B.5 (c) Using Register Built-ins - Storing and Manipulating Positions**

```
-----
---- Section 4-C: Execute PROG_1.TP and check the R[2]
-----
```

```

IF (status=cc_success) THEN --Verify SET_INT_REG is
    --successful
    WRITE('Executing "PROG_1.TP"',cr)
    CALL PROGLIN('PROG_1',1, prg_idx, FALSE)
--Execute PROG_1.TP starting on first line.
--Do not pause on entry of PROG_1.
    WRITE('Getting Value from R[2]',cr)
    GET_REG(2,r_flg, i_val, r_val, status) --Get R[2] value
    IF (status = cc_success) THEN --Verify GET_REG success
        IF (r_flg) THEN --REAL value in register
            WRITE('Got REAL value from R[2]',cr)
            IF (r_val <> 1.0) THEN --Verify value set by
                WRITE ('PROG_1 failed to set R[2]',cr)--PROG_1_TP
                WRITE ('PROG_1 failed to set R[2]',cr)
            ENDIF
        ELSE --Register contained an INTEGER
            WRITE('Got INTEGER value from R[2]',cr)
            IF (i_val <> 1) THEN --Verify value set by
                WRITE ('PROG_1 failed to set R[2]',cr) --PROG_1_TP
            ENDIF
        ENDIF
    ELSE --GET_REG was NOT successful
        WRITE('GET_REG Failed',cr, ' Status = ',status,cr)
    ENDIF
    ELSE --SET_INT_REG was NOT successful
        WRITE('SET_INT_REG Failed, Status = ',status,cr)
    ENDIF
    ELSE --SET_JPOS_REG was NOT successful
        WRITE('SET_JPOS_REG Failed, Status = ',status,cr)
    ENDIF
    ELSE --GET_POS_REG was NOT Successful
        WRITE('GET_POS_REG Failed, Status = ',status,cr)
    ENDIF
    ELSE --POS_REG_TYPE Failed, Status =
        WRITE ('POS_REG_TYPE Failed, Status = ', status, cr)
    ENDIF
    ELSE --SET_POS_REG was NOT successful
        WRITE('SET_POS_REG Failed, Status = ',status,cr)
    ENDIF
    IF (status = cc_success) THEN; WRITE ('Program Completed
Successfully',cr)
    ELSE ; WRITE ('Program Aborted due to
error',cr)
    ENDIF
END reg_ex

```

**Figure B.5 (d) Using Register Built-ins - Executing Program and Checking Register**

## B.6 POSITION DATA SET AND CONDITION HANDLERS PROGRAM

This program sets Position Data Set of the TP program, PTH\_SUB. And it calls PTH\_SUB. This example also sets up two global condition handlers.

- The first condition handler detects if the user has pushed a teach pendant key, and if so aborts the program.

- The second condition handler sets a variable when the program is aborted.

---- SET_POS.KL	
---- Section 0: Detail about SET_POS.kl	
---- Elements of KAREL Language Covered:	In Section:
---- Actions:	
---- ABORT	Sec 4-A
---- Clauses:	
---- WHEN	Sec 4-A
---- WITH	Sec 4-D
---- FROM	Sec 3-A
---- VIA	Sec 4-D
---- Conditions:	
---- ABORT	Sec 4-A
---- Data types:	
---- ARRAY OF REAL	Sec 2
---- BOOLEAN	Sec 2
---- INTEGER	Sec 2
---- JOINTPOS6	Sec 2
---- XYZWPR	Sec 2
---- Directives:	
---- ALPHABETIZE	Sec 1
---- COMMENT	Sec 1
---- ENVIRONMENT	Sec 1
---- Built-in Functions & Procedures:	
---- CHR	Sec 3-B; 4-B, D
---- CNV_REL_JPOS	Sec 4-D
---- SET_CURSOR	Sec 4-B
---- Statements:	
---- CONDITION...ENDCONDITION	Sec 4-A
---- FOR...ENDFOR	Sec 4-D
---- ROUTINE	Sec 3-A, B
---- WAIT FOR	Sec 3-B
---- WRITE	Sec 3-B; 4-B, C, D
---- Reserve Words:	
---- BEGIN	Sec 3-A, B, 4
---- CONST	Sec 2
---- END	Sec 3-A, B: 4-D
---- VAR	Sec 2
---- PROGRAM	Sec 1
---- Predefined File Names:	
---- TPFUNC	Sec 3-B; 4-D
---- TPDISPLAY	Sec 4-B

**Figure B.6 (a) Position Data Set and Condition Handlers Program - Overview**

```
---- Section 1: Program and Environment Declaration
----  

PROGRAM SET_POS      -- Define the program name
%ALPHABETIZE        -- Create the variables in alphabetical
                      -- order
%COMMENT            = 'Set TPE Position'
%ENVIRONMENT SYSDEF -- Necessary for using the $MOTYPE in
                      -- the MOVES
%ENVIRONMENT UIF    -- Necessary for SET_CURSOR
```

```
-----
----- Section 2: Constant and Variable Declarations
-----
CONST
    CH_ABORT    = 1           -- Number associated with the
                            -- abort Condition handler
    CH_F1        = 2           -- Number associated with the
                            -- F1 key Condition handler
VAR
    status       :INTEGER      -- Status from built-in calls
    prg_abrt    :BOOLEAN      -- Set when program is aborted
    strt_jnt    :JOINTPOS6   -- starting position of a move
    via_pos     :XYZWPR       -- via point for a circular move
    des_pos     :XYZWPR       -- destination point
    real_ary    :ARRAY[6] OF REAL -- This is used for
                                -- creating a joint
                                -- position with 6 axes
    index       :INTEGER      -- FOR loop counter
```

**Figure B.6 (b) Position Data Set and Condition Handlers Program - Declaration Section**

```
-----
----- Section 1: Program and Environment Declaration
-----
PROGRAM SET_POS          -- Define the program name
%ALPHABETIZE             -- Create the variables in alphabetical
                          -- order
%COMMENT      = 'Set TPE Position'
%ENVIRONMENT SYSDEF      -- Necessary for using the $MOTYPE in the
                          -- MOVES
%ENVIRONMENT UIF         -- Necessary for SET_CURSOR
-----
----- Section 2: Constant and Variable Declarations
-----
CONST
    CH_ABORT    = 1           -- Number associated with the
                            -- abort Condition handler
    CH_F1        = 2           -- Number associated with the
                            -- F1 key Condition handler
VAR
    status       :INTEGER      -- Status from built-in calls
    prg_abrt    :BOOLEAN      -- Set when program is aborted
    strt_jnt    :JOINTPOS6   -- starting position of a move
    via_pos     :XYZWPR       -- via point for a circular move
    des_pos     :XYZWPR       -- destination point
    real_ary    :ARRAY[6] OF REAL -- This is used for
                                -- creating a joint
                                -- position with 6 axes
    index       :INTEGER      -- FOR loop counter
```

**Figure B.6 (c) Position Data Set and Condition Handlers Program - Declare Routines**

```
-----
----- Section 4: Main Program
-----
BEGIN -- SET_POS
-----
----- Section 4-A: Global Condition Handler Declaration
-----
CONDITION[CH_ABORT]:
```

```

WHEN ABORT DO          -- When the program is aborting set prg_abrt
                      -- flag. This will be triggered if this
                      -- program aborts itself or if an external
                      -- mechanism aborts this program.
prg_abrt = TRUE        -- You may then have another task which detects
                      -- prg_abrt being set, and does shutdown
                      -- operations (ie: set DOUT/GOUT's, send
                      -- signals to a PLC)

ENDCONDITION

CONDITION[CH_F1]:
WHEN TPIN[129] DO      -- Monitor TP 'F1' Key. If 'F1' key is pressed,
ABORT                  -- abort the program.

ENDCONDITION

prg_abrt = false        -- Initialize variable which is set only
                        -- if the program is aborted and CH_ABORT
                        -- is enabled.

ENABLE CONDITION[CH_ABORT]  -- Start scanning abort condition as
                           -- defined.

ENABLE CONDITION[CH_F1]    -- Start scanning F1 key condition as
                           -- defined.

-----
----- Section 4-B: Display banner message and wait for users response
-----

TP_CLS                  -- Routine Call; Clears the TP USER
                        -- menu, and forces the TP USER menu
                        -- to be visible.

SET_CURSOR(TPDISPLAY,2,13, status)  -- Set cursor position in TP
                                    -- USER menu

IF (status <> 0 ) THEN      -- Verify that SET_CURSOR was successful
  WRITE ('SET_CURSOR built-in failed with status = ',status,cr)
  YES_NO                   -- Ask whether to quit, due to error.

ENDIF

--- Write heading in REVERSE video, then turn reverse video off
WRITE (chr(139),' PLEASE READ ',chr(143),CR)
WRITE (cr,' *** F1 Key is labelled as ABORT key *** ')
WRITE (cr,' Any time the F1 key is pressed the program')
WRITE (cr,' will abort. However, the F2 key is active ')
WRITE (cr,' only when the function key is labeled.',cr,cr)
YES_NO -- Wait for user response

```

**Figure B.6 (d) Position Data Set and Condition Handlers Program - Declare Condition Handlers**

```

-----
----- Section 4-D: Creating a joint position
-----

FOR indx = 1 to 6 DO          -- Set all joint angles to zero
  real_ary[indx] = 0.0
ENDFOR

real_ary[5] = 90.0            -- Make sure that the position
                             -- is not at a singularity point.

CNV_REL_JPOS(real_ary, strt_jnt, status)  -- Convert real_ary
                                             -- values into a joint
                                             -- position, strt_jnt

IF (status <> 0 ) THEN      -- Converting joint position
  -- was NOT successful
  WRITE ('CNV_REL_JPOS built-in failed with status = ',status,cr)
  YES_NO                   -- Ask user if want to continue.

ELSE                         -- Converting joint position was
                             -- successful.

  -- The start position, strt_jnt, has been created and is located

```

```

-- at axes 1-4 = 0.0, axes 5 = 90.0, axes 6 = 0.0.
via_pos = strt_jnt           -- Copy the strt_jnt to via_pos
via_pos.x = via_pos.x +200   -- Add offset to the x location
via_pos.y = via_pos.y +200   -- Add offset to the y location
-- The via position, via_pos, has been created to be the same
-- position as strt_jnt except it has been offset in the x and y
-- locations by 200 mm.
des_pos = strt_jnt           -- Copy the strt_jnt to des_pos
des_pos.x = des_pos.x + 400  -- Add offset to the x location
-- The destination position, des_pos, has been created to be the
-- same position as strt_jnt except it has been offset in the x
-- location by 400 mm.
OPEN_TPE('POS_SUB',TPE_RWACC, TPE_WRTREJ, open_id, STATUS)
IF STATUS <> 0 THEN
    WRITE (CR,'Open POS_SUB failed:',STATUS,CR)
    ABORT
ENDIF
SET_JPOS_TPE(open_id, 1, strt_jnt, STATUS,1)
IF STATUS <> 0 THEN
    WRITE (CR,'Write strt_pos to P[1] of POS_SUB
failed:',STATUS,CR)
    ABORT
ENDIF
SET_POS_TPE(open_id, 2, via_pos, STATUS,1)
IF STATUS <> 0 THEN
    WRITE (CR,'Write via_pos to P[2] of POS_SUB failed:',STATUS,CR)
    ABORT
ENDIF
SET_POS_TPE(open_id, 3, des_pos, STATUS,1)
WRITE (CR,'Moving to Destination Position',CR)
IF STATUS <> 0 THEN
    WRITE (CR,'Write des_pos to P[3] of POS_SUB failed:',STATUS,CR)
    ABORT
ENDIF
CLOSE_TPE(open_id, STATUS)
CALL_PROG('POS_SUB', prog_index)
ENDIF

WRITE ('SET_POS Successfully Completed',CR)
END SET_POS

```

**Figure B.6 (e) Position Data Set and Condition Handlers Program**

## B.7 LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS

This program displays the list of files on the FLPY: device, and lists the programs loaded on the controller. It also shows basic STRING manipulating capabilities, using semi-colons(;) as a statement separator, and nesting IF statements.

```

-----
---- LIST_EX.KL
-----
---- Section 0: Detail about LIST_EX.kl
-----
---- Elements of KAREL Language Covered: In Section:
---- Actions:
-----
```

```

---- Clauses:
----      FROM                               Sec 3-B
---- Conditions:
---- Data types:
----      ARRAY OF STRING                  Sec 2
----      BOOLEAN                         Sec 2, 3
----      INTEGER                         Sec 2, 3
----      STRING                          Sec 2
---- Directives:
----      %COMMENT                        Sec 1
----      %NOLOCKGROUP                   Sec 1
---- Built-in Functions & Procedures:
----      ABS                            Sec 4-A
----      ARRAY_LEN                      Sec 4-C&D
----      CNV_INT_STR                   Sec 4-A
----      FILE_LIST                     Sec 4-C
----      LOAD                           Sec 4-B
----      LOAD_STATUS                    Sec 4-B
----      PROG_LIST                     Sec 4-D
----      ROUND                          Sec 4-A
----      SUB_STR                       Sec 4-A
---- Statements:
----      FOR .... ENDFOR                 Sec 3-B
----      IF...THEN...ENDIF               Sec 4-A, B, C, D
----      ROUTINE                       Sec 3-A, B, C
----      REPEAT...UNTIL                Sec 4-C, D
----      RETURN                         Sec 3-A
----      WRITE                          Sec 3-B; 4-A, B
---- Reserve Words:
----      BEGIN                         Sec 3-A, B; 4
----      CONST                         Sec 2
----      CR                            Sec 3-B; 4-A, B
----      END                           Sec 3-A, B, 4-B
----      PROGRAM                       Sec 1
----      VAR                           Sec 2
---- Operators:
----      MOD                           Sec 3-A
----      /                             Sec 3-A
----      *                             Sec 3-A
---- Devices Used:
----      FLPY:                         Sec 4-C
---- Basic Concepts:
----      Semi-colon(;) as statement separator
----      Nested IF..THEN..ELSE..IF..THEN..ELSE..ENDIF..ENDIF
----      structure
----      Concatenation of STRINGS using '+'
---- 
```

**Figure B.7 (a) Listing Files and Programs and Manipulating Strings - Overview**

```

----- Section 1: Program and Environment Declaration
-----
PROGRAM LIST_EX
%NOLOCKGROUP    ---- Don't lock any motion groups
%COMMENT = 'FILE_LIST'
-----
---- Section 2: Constant and Variable Declarations

```

```

-----  

CONST  

    INCREMENT      = 13849  

    MODULUS        = 65536  

    MULTIPLIER     = 25173  

VAR  

    pr_cases       :STRING[6]   -- psuedo random number converted to  

                               -- string  

    prg_nm         :STRING[50]  -- Concatenated program name  

    loaded         :BOOLEAN    -- Used to see if program is loaded  

    initi          :BOOLEAN    -- Used to see if variables  

                               -- initialized  

    indx1          :INTEGER    -- FOR loop index  

    cases,          -- Random number returned  

    max_number,    -- Maximum random number  

    seed           :INTEGER    -- Seed for generating a random number  

    file_spec      :STRING[20]  -- File specification for FILE_LIST  

    n_files        :INTEGER    -- Number of files returned from  

                               -- FILE_LIST  

    n_skip         :INTEGER    -- Number to skip for FILE_LIST  

                               -- & PROG_LIST  

    format         :INTEGER    -- Format of returned names  

                               -- For FILE_LIST & PROG_LIST  

    ary_nam        :ARRAY[9] OF STRING[20] -- Returned names  

                               -- from FILE_LIST  

                               -- & PROG_LIST  

    prog_name      :STRING[10]  -- Program names to list from  

                               -- PROG_LIST  

    prog_type      :INTEGER    -- Program types to list from  

                               -- PROG_LIST  

    n_progs        :INTEGER    -- Number of programs returned from  

                               -- PROG_LIST  

    status          :INTEGER    -- Status of built-in procedure call
-----
```

**Figure B.7 (b) Listing Files and Programs and Manipulating Strings - Declarations Section**

```

-----  

----- Section 3: Routine Declaration  

-----  

-----  

----- Section 3-A: RANDOM Declaration  

-----           Creates a pseudo-random number and returns the  

-----           number.  

-----  

ROUTINE random(seed : INTEGER) : REAL  

BEGIN  

    seed = (seed * MULTIPLIER + INCREMENT) MOD MODULUS  

    RETURN(seed/65535.0)
END random  

-----  

----- Section 3-B: DISPL_LIST Declaration  

-----           Display maxnum elements of ary_nam.  

-----  

ROUTINE displ_list(maxnum :INTEGER)
BEGIN
    FOR indx1 = 1 TO maxnum DO ; WRITE (ary_nam[indx1],cr); ENDFOR
    -- Notice the use of the semi-colon, which allows multiple
    -- statements on a line.
END displ_list
-----
```

```
---- Section 3-C: TP_CLS Declaration
---- This routine is from ROUT_EX.KL and will
---- clear the TP USER menu screen and force it to be
---- visible.

ROUTINE tp_cls FROM rout_ex
```

**Figure B.7 (c) Listing Files and Programs and Manipulating Strings - Declare Routines**

```
---- Section 4: Main Program

BEGIN -- LIST_EX
tp_cls                                -- Use routine from the rout_ex.kl file

---- Section 4-A: Generate a pseudo random number, convert INTEGER
---- to STRING

max_number = 255 ;                      -- So the random number is 0..255
seed = 259 ;
WRITE ('Manipulating String',cr)
cases = ROUND(ABS((random(seed)*max_number)))-- Call random then
                                                -- take the absolute value
                                                -- of the number returned
                                                -- and round off the
                                                -- number.
CNV_INT_STR(cases, 1, 0, pr_cases)      -- Convert cases to its
                                           -- ascii representation
pr_cases = SUB_STR(pr_cases, 2,3)        -- get at most 3
                                           -- characters, starting
                                           -- at the second character,
                                           -- since first character
                                           -- is a blank.
```

**Figure B.7 (d) Listing Files and Programs and Manipulating Strings - Main Program**

```
---- Section 4-B: Build a program name from the number and try to
---- load it

--- Build a random program name to show the manipulation of
--- STRINGS and INTEGERS.
prg_nm = 'MYPROG' + pr_cases + '.PC'      -- Concatenate the STRINGS
                                              -- together which create a
                                              -- program name
--- Verify that the program is not already loaded
WRITE ('Checking load status of ',prg_nm,cr)
LOAD_STATUS(prg_nm, loaded, init)
IF (NOT loaded) THEN                      -- The program is not loaded
    WRITE ('Loading ',prg_nm,cr)
    LOAD(prg_nm, 0 , status)               -- Load in the program
    IF (status = 0 ) THEN                  -- Verify load is successful
        WRITE ('Loading ','MYPROG' + pr_cases + '.VR',cr)
        LOAD('MYPROG' + pr_cases + '.VR', 0, status) -- Load the .vr
                                                       -- file
        IF (status <> 0 ) THEN            -- Loading variables failed
            WRITE ('Loading of ', 'MYPROG' + pr_cases + '.VR', '
failed',cr)
            WRITE ('Status = ',status);
    ENDIF
```

```

        ELSE                                -- Load of program failed
            IF (status = 10003) THEN          -- File does not exist
                WRITE (prg_nm, ' file does not exist',cr)
            ELSE
                WRITE ('Loading of ',prg_nm, ' failed',cr,'Status =
',status);
            ENDIF
        ENDIF

        ELSE                                -- The program is already loaded
            IF (NOT initi) THEN           -- Variables not initialized
                WRITE ('Loading ','MYPROG' + pr_cases + '.VR',cr)
                LOAD('MYPROG' + pr_cases + '.VR', 0, status) -- Load in
                                                -- variables
                IF (status <> 0 ) THEN      -- Load of variables
                                                -- failed
                    WRITE ('Loading of ', 'MYPROG' + pr_cases + '.VR', '
failed',cr)
                    WRITE ('Status = ',status);
                ENDIF
            ENDIF
        ENDIF
    
```

**Figure B.7 (e) Listing Files and Programs and Manipulating Strings - Create and Load Program**

```

----- Section 4-C: Check the file listing of the drive FLPY: and
----- display them
-----
--- Display a directory listing of files on the Flpy:
file_spec = 'FLPY:.*.*'          -- All files in FLPY: drive
n_skip = 0                         -- First time do not skip any files
format = 3                          -- Return list in filename.filetype
format
WRITE ('Doing File list',cr)
REPEAT                            -- UNTIL all files have been listed
    FILE_LIST(file_spec, n_skip, format, ary_nam, n_files, status)
    IF (status <>0) THEN          -- Error occurred
        WRITE ('FILE_LIST built-in failed with Status = ',status,cr)
    ELSE
        displ_list(n_files)      -- Write the names to the TP USER menu
        n_skip = n_skip + n_files -- Skip the files we already got.
    ENDIF
UNTIL (ARRAY_LEN(ary_nam) <> n_files) -- When n_files does not
                                         -- equal declared size
                                         -- of ary_name then all
                                         -- files have been listed.
-----
----- Section 4-D: Show the programs loaded in controller
-----
--- Display the list of programs loaded on the controller
prog_name = '*'                  -- All program names should be listed
prog_type = 6                     -- Only PC type files should be listed
n_skip = 0                         -- First time do not skip any file
format = 2                          -- Return list in filename.filetype
format
WRITE ('Doing Program list',cr)
REPEAT                            -- UNTIL all programs have been listed
    PROG_LIST(prog_name, prog_type, n_skip, format, ary_nam,
n_progs, status)

```

```

-- The program names are stored in ary_nam
-- n_progs is the number of program names stored in ary_nam
IF (status <>0 ) THEN
    WRITE ('PROG_LIST built-in failed with Status = ',status,cr)
ELSE
    displ_list(n_progs)          -- Display the current list
    n_skip = n_skip + n_progs   -- Skip the programs already listed
ENDIF
UNTIL (ARRAY_LEN(ary_nam) <> n_progs) -- When n_files does not
                                         -- equal the declared size
                                         -- of ary_name then all
                                         -- programs have been
                                         -- listed.
END LIST_EX

```

**Figure B.7 (f) Listing Files and Programs and Manipulating Strings - List Programs**

## B.8 USING THE FILE AND DEVICE BUILT-INS

This program demonstrates how to use the File and Device built-ins. This program FORMATS and MOUNTS the RAM disk. Then copies files from the FLPY: device to RD:. If the RAM disk gets full the RAM disk size is increased and reformatted. This program continues until either all the files are copied successfully, or the built-in operations fail.

```

----- FILE_EX.KL -----
----- Section 0: Detail about FILE_EX.kl
----- Elements of KAREL Language Covered:           In Section:
----- Action:                                     Sec 1
----- Clauses:                                    Sec 1
----- FROM                                         Sec 3
----- Conditions:                                 Sec 1
----- Data types:
-----     BOOLEAN                                     Sec 2
-----     INTEGER                                     Sec 2
-----     STRING                                      Sec 2
----- Directives:
-----     COMMENT                                     Sec 1
-----     NOLOCKGROUP                                Sec 1
----- Built-in Functions & Procedures:
-----     CNV_TIME_STR                               Sec 4-A
-----     COPY_FILE                                  Sec 4-B
-----     DISMOUNT_DEV                                Sec 4-B
-----     FORMAT_DEV                                 Sec 4-B
-----     GET_TIME                                   Sec 4-A
-----     MOUNT_DEV                                  Sec 4-B
-----     PURGE_DEV                                 Sec 4-B
-----     SUB_STR                                    Sec 4-A
----- Statements:
-----     IF...THEN...ELSE...ENDIF                   Sec 4-B
-----     REPEAT...UNTIL                            Sec 4-A
-----     ROUTINE                                    Sec 3
-----     SELECT...ENDSELECT                         Sec 4-B
-----     WRITE                                     Sec 4-A, B

```

```
---- Reserve Words:
---- BEGIN Sec 4
---- CONST Sec 2
---- CR Sec 4-A,B
---- END Sec 4-B
---- PROGRAM Sec 4
---- VAR Sec 2
---- Devices Used:
---- FLPY Sec 4-B
---- MF3 Sec 4-B
---- RD Sec 4-B
---- FR Sec 4-B
```

**Figure B.8 (a) File and Device Built-ins Program - Overview**

```
-----  
---- Section 1: Program and Environment Declaration  
-----  
PROGRAM FILE_EX  
%nolockgroup  
%comment = 'COPY FILES'  
-----  
---- Section 2: Variable Declaration  
-----  
CONST  
    SUCCESS      = 0          -- Success status from built-ins  
    FINISHED     = TRUE       -- Finished Copy  
    TRY AGAIN   = FALSE      -- Try to copy again  
    RD FULL     = 85020     -- RAM disk full  
    NOT MOUNT   = 85005     -- Device not mounted  
    FR FULL     = 85001     -- FROM disk is full  
    MNT RD      = 85004     -- RAM disk must be mounted  
-- Refer to Error Code Manual (MARRUEROR02171E) or the OPERATOR'S  
-- MANUAL (Alarm Code List) (B-83284EN)  
  
VAR  
    time_int     : INTEGER  
    time_str     : STRING[30]  
    status        : INTEGER  
    cpy_stat     : BOOLEAN  
    to_dev        : STRING[5]  
-----  
---- Section 3: Routine Declaration  
-----  
ROUTINE tp_cls FROM ROUT_EX  
-----  
---- Section 4: Main program  
-----  
BEGIN -- FILE_EX  
    tp_cls      -- from rout_ex.kl  
-----  
---- Section 4-A: Get Time and FORMAT ramdisk with date as volume  
---- name  
-----  
    GET_TIME(time_int)           -- Get the system time  
    CNV_TIME_STR(time_int, time_str) -- Convert the INTEGER time  
                                    -- to readable format  
    WRITE ('Today is ', SUB_STR(time_str, 2,8),CR) -- Display the date  
                                    -- part
```

```
WRITE ('Time is ', SUB_STR(time_str, 11,5),CR) -- Display the time
-- part
```

**Figure B.8 (b) File and Device Built-ins Program - Declaration Section, Declare Routines**

```
-----
---- Section 4-B: Mount RAMDISK and start copying from FLPY to
---- MF3:
-----
to_dev = 'MF3:'                                -- Until all files have been copied
REPEAT
    cpy_stat = FINISHED
    WRITE('COPYing.....',cr)
    -- Copy the files from FLPY: to to_dev and overwrite the file if it
    -- already exists.
    COPY_FILE('FLPY:*.kl', to_dev, TRUE, FALSE, status)
    SELECT (status) OF
        CASE (RD_FULL):
            -- RAM disk is full
            -- Dismount and re-size the RAM-DISK
            WRITE ('DISMOUNTing RD: ....',cr)
            DISMOUNT_DEV('RD:', status)
                -- Verify DISMOUNT was successful or that
                -- the device was not already mounted
        IF (status = SUCCESS) OR (status = NOT_MOUNT) THEN
            -- Increase the size of RD:
            WRITE('Increasing RD: size...',cr)
            $FILE_MAXSEC = ROUND($FILE_MAXSEC * 1.2)
                -- Increase the RAM disk size
                -- Format the RAM-DISK
            WRITE('FORMATTING RD:.....',cr)
            FORMAT_DEV('RD:','',FALSE, status) -- Format the RAM disk
        IF (status <> SUCCESS) THEN
            WRITE ('FORMAT of RD: failed, status:', status,CR)
            WRITE ('Copy incomplete',cr)
        ELSE
            cpy_stat = TRY AGAIN
        ENDIF
        WRITE('MOUNTING RD:.....',cr)
        MOUNT_DEV ('RD:', status)
        IF (status <> SUCCESS) THEN
            WRITE ('MOUNTING of RD: failed, status:', status,CR)
            WRITE ('Copy incomplete',cr)
        ELSE
            cpy_stat = TRY AGAIN
        ENDIF
        ELSE
            WRITE ('DISMOUNT of RD: failed, status:', status,cr)
            WRITE ('Copy incomplete',cr)
        ENDIF
    CASE (FR_FULL):      -- FROM disk is full
        WRITE ('FROM disk is full',CR, 'PURGING FROM.....', CR)
        PURGE_DEV ('FR:', status) -- Purge the FROM
        IF (status <> SUCCESS) THEN
            WRITE ('PURGE of FROM failed, status:', status, CR)
            WRITE ('Copy incomplete', CR)
        ELSE
            cpy_stat = TRY AGAIN
        ENDIF
    CASE (NOT_MOUNT, MNT_RD): -- Device is not mounted
```

```

        WRITE ('MOUNTing ',to_dev,'.....',CR)
        MOUNT_DEV(to_dev, status)
        IF (status <> SUCCESS) THEN
            WRITE ('MOUNTing of ',to_dev,': failed, status:', status, CR)
            WRITE ('Copy incomplete', CR)
        ELSE
            cpy_stat = TRY AGAIN
        ENDIF
    CASE (SUCCESS):
        WRITE ('Copy completed successfully!',CR)
    ELSE:
        WRITE ('Copy failed, status:', status,CR)
    ENDSELECT
    UNTIL (cpy_stat = FINISHED)
END file_ex

```

**Figure B.8 (c) File and Device Built-ins Program - Mount and Copy to RAM Disk**

## B.9 USING DYNAMIC DISPLAY BUILT-INS

This program demonstrates how to use the dynamic display built-ins. This program initiates the dynamic display of various data types. It then executes another task, CHG\_DATA, which changes the values of these variables.

Before exiting this program the dynamic displays are canceled and the other task is aborted. If DYN\_DISP is aborted, it will set a variable which CHG\_DATA detects. This ensures that CHG\_DATA cannot continue executing once DYN\_DISP is aborted.

```

-----
----- DYN_DISP.KL
-----
----- Section 0: Detail about DYN_DISP.KL
-----
----- Elements of KAREL Language Covered:           In Section:
----- Actions:                                     Sec 3-C
----- Clauses:                                    Sec 2
-----          FROM                               Sec 4
-----          IN CMOS                            Sec 4
-----          WHEN
----- Conditions:                                Sec 4
-----          ABORT                             Sec 4
----- Data types:                                Sec 2
-----          BOOLEAN                           Sec 2
-----          INTEGER                           Sec 2
-----          REAL                              Sec 2
-----          STRING                            Sec 2
----- Directives:                                Sec 1
-----          ALPHABETIZE                      Sec 1
-----          COMMENT                           Sec 1
-----          NOLOCKGROUP                     Sec 1
-----
----- Built-in Functions & Procedures:
-----          ABORT_TASK                       Sec 4-C
-----          CNC_DYN_DISI                      Sec 4-C
-----          CNC_DYN_DISR                      Sec 4-C
-----          CNC_DYN_DISB                      Sec 4-C
-----          CNC_DYN_DISE                      Sec 4-C

```

-----	CNC_DYN_DISP	Sec 4-C
-----	CNC_DYN_DISS	Sec 4-C
-----	INI_DYN_DISI	Sec 3-A, 4-A
-----	INI_DYN_DISR	Sec 3-B, 4-A
-----	INI_DYN_DISB	Sec 3-C, 4-A
-----	INI_DYN_DISE	Sec 3-D, 4-A
-----	INI_DYN_DISP	Sec 3-E, 4-A
-----	INI_DYN_DISS	Sec 3-F, 4-A
-----	LOAD_STATUS	Sec 4-B
-----	LOAD	Sec 4-B
-----	RUN_TASK	Sec 4-B
-----		
-----	Statements:	
-----	CONDITION...ENDCONDITION	Sec 4
-----	IF...THEN...ENDIF	Sec 4-A, B, C
-----	READ	Sec 4-C
-----	ROUTINE	Sec 3-A, B, C
-----	WRITE	Sec 4-A, B, C
-----		
-----	Reserve Words:	
-----	BEGIN	Sec 3-A, B; 4
-----	CR	Sec 4-A
-----	CONST	Sec 2
-----	END	Sec 3-A, B; 4-C
-----	PROGRAM	Sec 4
-----	VAR	Sec 2
-----	Predefined File Variables:	
-----	TPPPROMPT	Sec 4-B, C
-----	Predefined Windows:	
-----	T_FU	Sec 3-A, B

**Figure B.9 (a) Using Dynamic Display Built-ins - Overview**

```
-----  
----- Section 1: Program and Environment Declaration  
-----  
PROGRAM DYN_DISP  
%nolockgroup  
%comment = 'Dynamic Disp'  
%alphabetize  
%INCLUDE KLIOTYPS  
-----  
----- Section 2: Variable Declaration  
-----  
CONST  
  cc_success      = 0      -- Success status  
  cc_clear_win    = 128     -- Clear window  
  cc_clear_eol    = 129     -- Clear to end of line  
  cc_clear_eow    = 130     -- Clear to end of window  
  CH_ABORT        = 1      -- Condition Handler to detect when  
                           -- program aborts  
VAR  
  Int_wind         :STRING[10]  
  Rel_wind         :STRING[10]  
  Field_Width      :INTEGER  
  Attr_Mask        :INTEGER  
  Char_Size        :INTEGER  
  Row              :INTEGER  
  Col              :INTEGER  
  Interval         :INTEGER
```

```

Buffer_Size      :INTEGER
Format          :STRING[7]
bool_names      :ARRAY[2] OF STRING[10]
enum_names      :ARRAY[4] OF STRING[10]
pval_names      :ARRAY[2] OF STRING[10]
bool1 IN CMOS   :BOOLEAN
enum1 IN CMOS   :INTEGER
port_type        :INTEGER
port_no          :INTEGER
Str1 IN CMOS    :STRING[10]
Int1 IN CMOS    :INTEGER      -- Using IN CMOS will create the
                                -- variables
Real1 IN CMOS   :REAL         -- in CMOS RAM, which is permanent
                                -- memory.
status           :INTEGER
loaded,          :
initialized     :BOOLEAN
dynd_abrt       :BOOLEAN      -- Set to true when program aborts.

```

**Figure B.9 (b) Using Dynamic Display Built-ins - Declaration Section**

```

----- Section 3: Routine Declaration
-----
----- Section 3-A: SET_INT Declaration
----- Set all the input parameters for the
----- INIT_DYN_DISI call.
-----
ROUTINE Set_Int
Begin
  -- Valid predefined windows are described in
  -- Chapter 7.9.1, "USER MENU on the Teach Pendant
  -- Error Line      --> 'ERR'      1 line
  -- Status Line     --> 'T_ST'     3 lines
  -- Display Window  --> 'T_FU'     10 lines
  -- Prompt Line    --> 'T_PR'     1 line
  -- Function Key   --> 'T_FK'     1 line
  Int_Wind          = 'T_FU'      -- Use the predefined display
                                  -- window
  Field_Width       = 0          -- Use the minimum width
                                  -- necessary
  Attr_Mask         = 1 OR 4     -- BOLD and UNDERLINED
  Char_Size          = 0          -- Normal
  Row                = 1          -- Specify the location within 'T_FU'
  Col                = 16         -- to dynamically displayed
  Interval           = 250        -- 250ms between updates
  Buffer_Size        = 10         -- Minimum value required.
  Format             = '%-8d'     -- 8 character minimum field width
  --- With this specification the INTEGER will be displayed as follows:
  ---           -----
  ---           |xxxxxxxxx|
  ---           -----
  ---           Where the integer value will be left justified.
  ---           The x's will be the integer value unless the integer value
  ---           is less than 8 characters, then the right side will be
  ---           blanks up to a total 8 characters. If the integer value is
  ---           greater than the 8 characters the width is dynamically
  ---           increased to display the whole integer value. The INTEGER
  ---           value will also be bold and underlined.

```

```

End Set_Int
-----
---- Section 3-B: SET_REAL Declaration
---- Set all the input parameters for the
----INI_DYN_DISR call.
-----

ROUTINE Set_Real
Begin
    Rel_Wind      = 'T_FU' -- Use the predefined display
                         -- window
    Field_Width   = 10    -- Maximum width of display.
    Attr_Mask     = 2 OR 8 -- blinking and reverse video
    Char_Size     = 0     -- Normal
    Row           = 2     -- Specify the location within
    Col           = 16    -- 'TFU' to dynamically display
    Interval      = 200   -- 200ms between update
    Buffer_Size   = 10    -- Minimum value required.
    Format        = '%2.2f'
--- With the format and field_width specification the REAL will be
--- displayed as follows:
---      -----
---      |xxxx.xx  |
---      -----
---      Where the real value will be left justified.
---      There will always be two digits after the decimal point.
---      A maximum width of 10 will be used.
---      If the real value is less than 10 characters the right side
---      will be padded with blanks up to 10 character width.
---      If the real value exceeds 10 characters, the display width
---      will not expand but will display a ">" as the last
---      character, indicating the entire value is not displayed.
---      The value will also be blinking and in reverse video.
End Set_Real
-----
---- Section 3-C: SET_BOOL Declaration
---- Set all the input parameters for the
----INI_DYN_DISB call
-----

ROUTINE Set_Bool
Begin
    -- Valid predefined windows are described in
    -- Chapter 7.9.1, "USER MENU on the Teach Pendant
    -- Error Line    --> 'ERR'    1 line
    -- Status Line   --> 'T_ST'   3 lines
    -- Display Window --> 'T_FU'   10 lines
    -- Prompt Line   --> 'T_PR'   1 line
    -- Function Key  --> 'T_FK'   1 line
    Int_Wind       = 'T_FU' -- Use the predefined display
                           -- window
    Field_Width   = 10    -- Display 10 chars
    Attr_Mask     = 2     -- Blinking
    Char_Size     = 0     -- Normal
    Row           = 3     -- Specify the location within
    Col           = 16    -- 'T_FU' to dynamically displayed
    Interval      = 250   -- 250ms between updates
    Buffer_Size   = 10    -- Minimum value required
    bool_names[1]  = 'YES' -- string display in bool_var is
                           -- FALSE
    bool_names[2]  = 'NO'  -- string display in bool_var is
                           -- TRUE
--- With this specification the BOOLEAN will be displayed as follows:

```

```

---          -----
---          |xxxxxxxxx|
---
---      Where the boolean value will be left justified.
---      The x's will be one of the strings 'YES' or 'NO', depending
---      on the value of booll.  The string will be blinking.
End Set_Boolean

-----
---      Section 3-D: SET_ENUM Declaration
---          Set all the input parameters for the
---         INI_DYN_DISE call.

-----
ROUTINE Set_Enum
Begin
    -- Valid predefined windows are described in
    -- Chapter 7.9.1, "USER MENU on the Teach Pendant
    -- Error Line    --> 'ERR'    1   line
    -- Status Line   --> 'T_ST'   3   lines
    -- Display Window --> 'T_FU'   10  lines
    -- Prompt Line   --> 'T_PR'   1   line
    -- Function Key  --> 'T_FK'   1   line
    Int_Wind        = 'T_FU' -- Use the predefined display
                           -- window
    Attr_Mask       = 8       -- REVERSED
    Field_Width    = 10      -- Display to characters
    Char_Size       = 0       -- Normal
    Row             = 4       -- Specify the location within
    Col             = 16      -- 'T_FU' to dynamically displayed
    Interval        = 250     -- 250ms between updates
    Buffer_Size     = 10      -- Minimum value required
    enum_names[1]   = 'Enum-0' -- value displayed if
                           -- enum_var = 0
    enum_names[2]   = 'Enum-1' -- value displayed if
                           -- enum_var = 1
    enum_names[3]   = 'Enum-2' -- value displayed if
                           -- enum_var = 2
    enum_names[4]   = 'Enum-3' -- value displayed if
                           -- enum_var = 3
    --- With this specification enum_var will be displayed as follows:
    ---          -----
    ---          |xxxxxxxxx|
    ---
    ---      Where one of the strings enum_names will be displayed,
    ---      depending on the integer value enum1.  If enum1 is outside
    ---      the range 0-3, a string of 10 '?'s will be displayed.
    ---      The string will be displayed in reversed video.
End Set_Enum

-----
---      Section 3-E: SET_PORT Declaration
---          Set all the input parameters for the
---         INI_DYN_DISP call.

-----
ROUTINE Set_Port
Begin
    -- Valid predefined windows are described in
    -- Chapter 7.9.1, "USER MENU on the Teach Pendant
    -- Error Line    --> 'ERR'    1   line
    -- Status Line   --> 'T_ST'   3   lines
    -- Display Window --> 'T_FU'   10  lines
    -- Prompt Line   --> 'T_PR'   1   line
    -- Function Key  --> 'T_FK'   1   line

```

```

Int_Wind          = 'T_FU'    -- Use the predefined display
                    -- window
Field_Width       = 10       -- Display to characters
Attr_Mask         = 1        -- BOLD
Char_Size         = 0        -- Normal
Row               = 5        -- Specify the location within
Col               = 16       -- 'T_FU' to dynamically displayed
Interval          = 250      -- 250ms between updates
Buffer_Size        = 10       -- Minimum value required.
pval_names[1]     = 'RELEASED' -- text displayed if key is not
                            -- pressed
pval_names[2]     = 'PRESSED'  -- text displayed if key is
                            -- pressed
port_type         = io_tpin   -- port type = TP key
port_no           = 175      -- user-key 3
--- With this specification PRESSED or RELEASED will be displayed as
--- follows:
---             -----
---             |xxxxxxxxx|
---             -----
---             Where the string will be left justified.
---             The x's will be either 'RELEASED' or 'PRESSED'.
---             The string will also be normal video.
---             (Bold is not supported on the teach pendant.)
End Set_Port
-----
---- Section 3-F: SET_STR Declaration
---- Set all the input parameters for the
INI_DYN_DISS call.
-----
ROUTINE Set_Str
Begin
  -- Valid predefined windows are described in
  -- Chapter 7.9.1, "USER MENU on the Teach Pendant
  -- Error Line    --> 'ERR'    1 line
  -- Status Line   --> 'T_ST'   3 lines
  -- Display Window --> 'T_FU'  10 lines
  -- Prompt Line   --> 'T_PR'   1 line
  -- Function Key  --> 'T_FK'   1 line
  Int_Wind         = 'T_FU'    -- Use the predefined display
                                -- window
  Field_Width       = 10       -- Use the minimum width necessary
  Attr_Mask         = 1 OR 4   -- BOLD and UNDERLINED
  Char_Size         = 0        -- Normal
  Row               = 6        -- Specify the location within
  Col               = 16       -- 'T_FU' to dynamically displayed
  Interval          = 250      -- 250ms between updates
  Buffer_Size        = 10       -- Minimum value required.
  Format            = '%10s'   -- 10 character minimum field
                                -- width
--- With this specification the STRING will be displayed as follows:
---             -----
---             |xxxxxxxxx|
---             -----
---             Where the string value will be left justified.
---             The x's will be the string value.
---             The STRING will also be underlined.
End Set_Str
-----
---- Section 3-G: TP_CLS Declaration
---- Clear the TP USER menus screen and force it to be

```

```
---- visible.
-----
ROUTINE tp_cls FROM rout_ex
```

**Figure B.9 (c) Using Dynamic Display Built-ins - Declare Routines**

```
-----  
----- Section 4: Main program  
-----  
BEGIN --- DYN_DISP  
dynd_abrt = FALSE  
CONDITION [CH_ABORT]:  
    WHEN ABORT DO -- When the program is aborting set dynd_abrt flag.  
        -- This will be triggered if this program aborts itself  
        -- or if an external mechanism aborts this program.  
        dynd_abrt = TRUE -- CHG_DATA will detect this and complete  
            -- execution.  
ENDCONDITION  
ENABLE CONDITION [CH_ABORT]  
-----  
----- Section 4-A: Setup variables, initiate dynamic display  
-----  
TP_CLS           -- Clear the TP USER screen  
                  -- Force display of the TP USER screen  
STATUS = cc_success -- Initialize the status variable  
-- Initialize the dynamically displayed variables  
Int1      = 1  
Real1     = 1.0  
Bool1     = FALSE  
Enum1     = 0  
Str1      = ''  
-- Display messages to the TP USER screen  
WRITE ('Current INT1 =',CR)  
WRITE ('Current REAL1=',CR)  
WRITE ('Current BOOL1=',CR)  
WRITE ('Current ENUM1=',CR)  
WRITE ('Current PORT =',CR)  
WRITE ('Current STR1 =',CR)  
Set_Int   -- Set parameter values for INTEGER DYNAMIC DISPLAY  
INI_DYN_DISI(Int1,Int_Wind,Field_Width,Attr_Mask,Char_Size,  
             Row,Col, Interval, Buffer_Size, Format ,Status)  
IF Status <> cc_success THEN -- Check the status  
    WRITE('INI_DYN_DISI failed, Status=',status,CR)  
ENDIF  
Set_Bool -- Set parameter values for BOOLEAN DYNAMIC DISPLAY  
INI_DYN_DISB(Bool1,Int_Wind,Field_Width,Attr_Mask,Char_Size,  
             Row,Col, Interval, bool_names,Status)  
IF Status <> cc_success THEN -- Check the status  
    WRITE('INI_DYN_DISB failed, Status=',status,CR)  
ENDIF  
Set_Enum -- Set parameter values for Enumerated Integer  
          -- DYNAMIC DISPLAY  
INI_DYN_DISE(Enum1,Int_Wind,Field_Width,Attr_Mask,Char_Size,  
             Row,Col, Interval, enum_names,Status)  
IF Status <> cc_success THEN -- Check the status  
    WRITE('INI_DYN_DISE failed, Status=',status,CR)  
ENDIF  
Set_Port -- Set parameter values for Port DYNAMIC DISPLAY  
INI_DYN_DISP(port_type,  
            port_no ,Int_Wind,Field_Width,Attr_Mask,Char_Size,
```

```

        Row,Col, Interval, pval_names, Status)
IF Status <> cc_success THEN -- Check the status
    WRITE('INI_DYN_DISP failed, Status=',status,CR)
ENDIF
Set_Real -- Set parameter values for REAL DYNAMIC DISPLAY
INI_DYN_DISR(Reall,Rel_Wind,Field_Width,Attr_Mask,Char_Size,
              Row,Col, Interval, Buffer_Size, Format ,Status)
IF Status <> cc_success THEN -- Check the status
    WRITE('INI_DYN_DISR failed, Status=',status,CR)
ENDIF
Set_Str -- Set parameter values for STRING DYNAMIC DISPLAY
INI_DYN_DISS(Str1,Int_Wind,Field_Width,Attr_Mask,Char_Size,
              Row,Col, Interval, Buffer_Size, Format ,Status)
IF Status <> cc_success THEN -- Check the status
    WRITE('INI_DYN_DISS failed, Status=',status,CR)
ENDIF

```

**Figure B.9 (d) Using Dynamic Display Built-ins - Initiate Dynamic Displays**

```

----- Section 4-B: Check on subordinate program and execute it.
-----
-- Check the status of the other program which will change the
-- value of the variables.
LOAD_STATUS('chg_data', loaded, initialized)
IF (loaded = FALSE ) THEN
    WRITE TPPROMPT(CHR(cc_clear_win)) -- Clear the prompt line
    WRITE TPPROMPT('CHG_DATA is not loaded. Loading now... ')
    LOAD('chg_data.pc',0,status)
    IF (status = cc_success) THEN -- Check the status
        RUN_TASK('CHG_DATA',1,false,false,1,status)
        IF (Status <> cc_success) THEN -- Check the status
            WRITE ('Changing the value of the variables',CR)
            WRITE ('by another program failed',CR)
            WRITE ('BUT you can try changing the values',CR)
            WRITE ('from KCL',CR)
        ENDIF
    ELSE
        WRITE ('LOAD Failed, status = ',status,CR)
    ENDIF
ELSE
    RUN_TASK('CHG_DATA',1,false,false,1,status)
    IF (Status <> cc_success) THEN -- Check the status
        WRITE ('Changing the value of the variables',CR)
        WRITE ('by another program failed',CR)
        WRITE ('BUT you can try changing the values',CR)
        WRITE ('from KCL',CR)
    ENDIF
ENDIF

```

**Figure B.9 (e) Using Dynamic Display Built-ins - Execute Subordinate Task**

```

----- Section 4-C: Wait for user response, and cancel dynamic
----- displays
-----
WRITE TPPROMPT(CHR(cc_clear_win)) -- Clear the prompt line
WRITE TPPROMPT('Enter a number to cancel DYNAMIC display: ')
READ (CR) -- Read only one character
-- See Chapter 7.7.1,

```

```

-- "Formatting INTEGER Data Items"
ABORT_TASK('CHG_DATA',TRUE, TRUE,status) -- Abort CHG_DATA
IF (status <> cc_success) THEN -- Check the status
    WRITE(' ABORT_TASK failed, Status=',status,CR)
ENDIF
CNC_DYN_DISI(Int1, Int_Wind,Status) -- Cancel display of Int1
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISI failed, Status=',status,CR)
ENDIF
CNC_DYN_DISR(Reall1, Rel_Wind,Status) -- Cancel display of Reall1
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISR failed, Status=',status,CR)
ENDIF
CNC_DYN_DISB(Bool1, Int_Wind,Status) -- Cancel display of Bool1
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISB failed, Status=',status,CR)
ENDIF
CNC_DYN_DISE(Enum1, Int_Wind,Status) -- Cancel display of Enum1
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISE failed, Status=',status,CR)
ENDIF
CNC_DYN_DISP(port_type, Port_no, Int_Wind,Status) -- Cancel display
-- of Port
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISP failed, Status=',status,CR)
ENDIF
CNC_DYN_DISS(Str1, Int_Wind,Status) -- Cancel display of String
IF Status <> 0 THEN -- Check the status
    WRITE(' CND_DYN_DISS failed, Status=',status,CR)
ENDIF
END DYN_DISP

```

**Figure B.9 (f) Using Dynamic Display Built-ins - User Response Cancels Dynamic Displays**

## B.10 MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES

The CHG\_DATA.KL program is called by DYN\_DISP, where the actual variables are displayed dynamically. This program does some processing and changes the value of those variables.

```

-----
---- CHG_DATA.KL
-----

-----
---- Section 0: Detail about CHG_DATA.KL
-----

---- Elements of KAREL Language Covered:           In Section:
---- Actions:                                     Sec 2
---- Clauses:                                     Sec 2
----          FROM
---- Conditions:                                 Sec 2
---- Data types:                                Sec 2
----          INTEGER
----          REAL
---- Directives:
---- Built-in Functions & Procedures:

```

```
---- Statements:
----          DELAY                      Sec 4
----          FOR....ENDFOR              Sec 4
----          REPEAT...UNTIL            Sec 4
----          -
---- Reserve Words:
----          BEGIN                     Sec 4
----          END                       Sec 4
----          PROGRAM                   Sec 1
----          VAR                       Sec 2
----          -
----          Section 1: Program and Environment Declaration
----          -
PROGRAM CHG_DATA
%nolockgroup
%comment = 'Dynamic Disp2'
```

**Figure B.10 (a) Manipulate Dynamically Displayed Variables - Overview**

```
----          Section 2: Variable Declaration
----          -
VAR
-- IF the following variables did NOT have IN CMOS, the following
-- errors would be posted when loading this program:
--   VARS-012 Create var -INT1 failed   VARS-038 Cannot change
--   CMOS/DRAM type
--   VARS-012 Create var -REAL1 failed VARS-038 Cannot change
--   CMOS/DRAM type
-- This indicates that there is a discrepancy between DYN_DISP and
-- CHG_DATA.
-- One program has specified to create the variables in DRAM the
-- other specified CMOS.
  Int1 IN CMOS FROM dyn_disp :INTEGER      -- dynamically displayed
                                         -- variable
  Real1 IN CMOS FROM dyn_disp :REAL        -- dynamically displayed
                                         -- variable
  Booll IN CMOS FROM dyn_disp :BOOLEAN     -- dynamically displayed
                                         -- variable
  Enum1 IN CMOS FROM dyn_disp :INTEGER     -- dynamically displayed
                                         -- variable
  Str1 IN CMOS FROM dyn_disp :STRING[10]   -- dynamically displayed
                                         -- variable
  indx :INTEGER
  dynd_abrt FROM dyn_disp :BOOLEAN         -- Set in dyn_disp when
                                         -- dyn_disp is aborting
----          Section 3: Routine Declaration
----          -
```

**Figure B.10 (b) Manipulate Dynamically Displayed Variables - Declaration Section**

```
----          Section 4: Main program
----          -
BEGIN  -- CHG_DATA
-- This demonstrates that the variables are changed from this task,
-- CHG_DATA.
```

```

-- The dynamic display initiated in task DYN_DISP, will continue
-- to correctly display the updated values of these variables.
-- Do real application processing.
-- Simulated here in a FOR loop.
REPEAT
  FOR indx = -9999 to 9999 DO
    int1 = (indx DIV 2) * 7
    real1 = (indx DIV 3)* 3.13
    bool1 = ((indx AND 4) = 0)
    enum1 = (ABS(indx) DIV 5) MOD 5
    Str1 = SUB_STR('123456789A', 1, (ABS(indx) DIV 6) MOD 7 + 1)
    delay 200 -- Delay for 1/5 of a second as if processing is
               -- going on.
  ENDFOR
UNTIL (DYND_ABRT) -- This task is aborted from DYN_DISP. However,
                   -- if DYN_DISP aborts abnormally (ie from a
                   -- KCL> ABORT), it will set DYND_ABRT, which
                   -- will allow CHG_DATA to complete execution.
END CHG_DATA

```

**Figure B.10 (c) Manipulate Dynamically Displayed Variables - Main Program**

## B.11 DISPLAYING A LIST FROM A DICTIONARY FILE

This program controls the display of a list which is read in from the dictionary file DCLISTEG.UTX. For more information on DCLISTEG.UTX refer to [Section B.11.1, Dictionary Files](#). DCLST\_EX.KL controls the placement of the cursor along with the action taken for each command.

```

----- DCLST_EX.KL
-----
----- Section 0: Detail about DCLST_EX.KL
-----
----- Elements of KAREL Language Covered: In Section:
----- Actions:
----- Clauses:
-----      FROM                      Sec 3-E
----- Conditions:
----- Data types:
-----      ARRAY OF STRING          Sec 2
-----      BOOLEAN                  Sec 2
-----      DISP_DAT_T              Sec 2
-----      FILE                     Sec 2
-----      INTEGER                  Sec 2
-----      STRING                   Sec 2
----- Directives:
-----      ALPHABETIZE             Sec 1
-----      COMMENT                 Sec 1
-----      INCLUDE                 Sec 1
-----      NOLOCKGROUP            Sec 1
----- Built-in Functions & Procedures:
-----      ADD_DICT                Sec 3-B
-----      ACT_SCREEN              Sec 4-I
-----      ATT_WINDOW_S            Sec 4-C
-----      CHECK_DICT              Sec 3-B
-----      CLR_IO_STAT             Sec 3-A,C
-----      CNV_STR_INT              Sec 4-E

```

----	DEF_SCREEN	Sec 4-B
----	DET_WINDOW	Sec 4-I
----	DISCTRL_LIST	Sec 4-G, H
----	FORCE_SPMENU	Sec 4-A, B, C
----	IO_STATUS	Sec 3-A,
----	ORD	Sec 4-H
----	READ_DICT	Sec 4-E, F
----	REMOVE_DICT	Sec 4-I
----	SET_FILE_ATR	Sec 4-G
----	STR_LEN	Sec 4-F
----	UNINIT	Sec 3-C
----	WRITE_DICT	Sec 4-D, H, I
<b>Statements:</b>		
----	ABORT	Sec 3-B
----	CLOSE FILE	Sec 4-I
----	FOR...ENDFOR	Sec 4-F
----	IF...THEN...ENDIF	Sec 3-A, B, C, D; 4-F, H, I
----	OPEN FILE	Sec 3-A; 4-H
----	READ	Sec 4-A, B, H
----	REPEAT...UNTIL	Sec 4-H
----	ROUTINE	Sec 3-A, B, C, D, E
----	SELECT...ENDSELECT	Sec 4-H
----	WRITE	Sec 3-A, B, C, D; 4-A, I
<b>Reserve Words:</b>		
----	BEGIN	Sec 3-A, B, C, D; 4
----	CR	Sec 4-A, B, C
----	END	Sec 3-A, B, C, D; 4-I
----	PROGRAM	Sec 1
----	VAR	Sec 2
<b>Predefined File Names:</b>		
----	TPDISPLAY	Sec 4-D, G, H, I
----	TPFUNC	Sec 4-D, H
----	TPPROMPT	Sec 4-D, H, I
----	TPSTATUS	Sec 4-D, I
<b>Devices Used:</b>		
----	RD2U	Sec 3-B
<b>Predefined Windows:</b>		
----	ERR	Sec 4-C
----	T_ST	Sec 4-C
----	T_FU	Sec 4-C
----	T_PR	Sec 4-C
----	T_FR	Sec 4-C

**Figure B.11 (a) Display List from Dictionary File - Overview**

```
-----  
---- Section 1: Program and Environment Declaration  
-----  
PROGRAM DCLST_EX  
%COMMENT='DISCTRL_LIST '  
%ALPHABETIZE  
%NOLOCKGROUP  
%INCLUDE DCLIST -- the include file from the dictionary DCLISTEG.UTX  
-----  
---- Section 2: Variable Declarations  
-----  
VAR  
  exit_Cmnd      : INTEGER  
  act_pending    : INTEGER      -- decide if any action is pending  
  display_data   : DISP_DAT_T  -- information needed for DISCTRL_LIST
```

```

done          : BOOLEAN      -- decides when to complete execution
Kb_file       : FILE         -- file opened to the TPKB
i             : INTEGER      -- just a counter
key           : INTEGER      -- which key was pressed
last_line     : INTEGER      -- number of last line of information
list_data     : ARRAY[20] OF STRING[40] -- exact string
                           -- information
num_options   : INTEGER      -- number of items in list
old_screen    : STRING[4]   -- previously attached screen
status         : INTEGER      -- status returned from built-in call
str            : STRING[1]   -- string read in from teach pendant
Err_file      : FILE         -- err log file
Opened         : BOOLEAN      -- err log file open or not

```

**Figure B.11 (b) Display List from Dictionary File - Declaration Section**

```

-----  

---- Section 3: Routine Declaration  

-----  

-----  

---- Section 3-A: Op_Err_File Declaration  

---- Open the error log file.  

-----  

Routine Op_Err_File  

Begin  

  Opened = false  

  Write TPPROMPT(CR, 'Creating Auto Error File .....')  

  OPEN FILE Err_File ('RW','RD2U:\D_LIST.ERR') -- open for output  

  IF (IO_STATUS(Err_File) <> 0 ) THEN  

    CLR_IO_STAT(Err_File)  

    Write TPPROMPT('*** USE USER WINDOW FOR ERROR OUTPUT ***',CR)  

  ELSE  

    Opened = TRUE  

  ENDIF  

End Op_Err_File  

-----  

---- Section 3-B: Chk_Add_Dct Declaration  

---- Check whether a dictionary is loaded.  

---- If not loaded then load in the dictionary.  

-----  

Routine Chk_Add_Dct  

Begin -- Chk_Add_Dct  

  -- Make sure 'DLST' dictionary is added.  

  CHECK_DICT('DLST',TPTSSP_TITLE,STATUS)  

  IF STATUS <> 0 THEN  

    Write TPPROMPT(CR, 'Loading Required Dictionary.....')  

    ADD_DICT('RD2U:\DCLISTEG','DLST',DP_DEFAULT,DP_DRAM,STATUS)  

    IF status <> 0 THEN  

      WRITE TPPROMPT('ADD_DICT failed, STATUS=',STATUS,CR)  

      ABORT -- Without the dictionary this program can not  

             -- continue.  

    ENDIF  

  ELSE  

    WRITE TPPROMPT ('Dictionary already loaded in system. ')  


```

```

        ENDIF
End Chk_Add_Dct

```

**Figure B.11 (c) Display List from Dictionary File - Declare Routines**

```

-----  

---- Section 3-C: Log_Errors Declaration  

---- Log detected errors to a file to be reviewed  

---- later.  

-----  

ROUTINE Log_Errors(Out: FILE; Err_Str:STRING;Err_No:INTEGER)  

BEGIN  

    IF NOT Opened THEN -- If error log file not opened then write  

        -- errors to screen  

        WRITE (Err_Str,Err_No,CR)  

    ELSE  

        IF NOT UNINIT(Out) THEN  

            CLR_IO_STAT(Out)  

            WRITE Out(Err_Str,Err_No,CR,CR)  

        ELSE  

            WRITE (Err_Str,Err_No, CR)  

        ENDIF  

    ENDIF  

END Log_Errors  

-----  

---- Section 3-D: Chk_Stat Declaration  

---- Check the global variable, status.  

---- If not zero then log input parameter, err_str,  

---- to error file.  

-----  

ROUTINE Chk_Stat ( err_str: STRING)  

BEGIN -- Chk_Stat  

    IF( status <> 0) then  

        Log_Errors(Err_File, err_str,Status)  

    ENDIF  

END Chk_Stat  

-----  

---- Section 3-E: TP_CLS Declaration  

-----  

ROUTINE TP_CLS FROM ROUT_EX

```

**Figure B.11 (d) Display List from Dictionary File - Declare Error Routines**

```

-----  

---- Section 4: Main Program  

-----  

BEGIN -- DCLST_EX  

-----  

---- Section 4-A: Perform Setup operations  

-----  

    TP_CLS -- Call routine to clear and force the TP USER menu to be  

        -- visible  

    Write ('***** Starting DISCTRL_LIST Example *****', CR, CR)  

    Chk_Add_Dct -- Call routine to check and add dictionary  

    Op_Err_File -- Call routine open error log file  

-----  

---- Section 4-B: Define and Active a screen  

-----  

    DEF_SCREEN('LIST', 'TP', status) -- Create/Define a screen  

        -- called LIST

```

```

Chk_Stat ('DEF_SCREEN LIST')      -- Verify DEF_SCREEN was
                                  -- successful
ACT_SCREEN('LIST', old_screen, status) -- activate the LIST screen
                                      -- that was just defined.
Chk_Stat ('ACT_SCREEN LIST')-- Verify ACT_SCREEN was successful
-----
----- Section 4-C: Attach windows to the screen
-----
-- Attach the required windows to the LIST screen.
-- SEE:
-- Chapter 7.9.1 "USER Menu on the Teach Pendant,
-- for more details on predefined window names.
ATT_WINDOW_S('ERR', 'LIST', 1, 1, status) -- attach the error
                                            -- window
    Chk_Stat('Attaching ERR')
ATT_WINDOW_S('T_ST', 'LIST', 2, 1, status) -- attach the status
                                            -- window
    Chk_Stat('T_ST not attached')
ATT_WINDOW_S('T_FU', 'LIST', 5, 1, status) -- attach the full
                                            -- window
    Chk_Stat('T_FU not attached')
ATT_WINDOW_S('T_PR', 'LIST', 15, 1, status)-- attach the prompt
                                            -- window
    Chk_Stat('T_PR not attached')
ATT_WINDOW_S('T_FK', 'LIST', 16, 1, status)-- attach the function
                                            -- window
    Chk_Stat('T_FK not attached')

```

**Figure B.11 (e) Display List from Dictionary File - Setup and Define Screen**

```

----- Section 4-D: Write dictionary elements to windows
-----
-- Write dictionary element,TPTSSP_TITLE, from DLST dictionary.
-- Which will clear the status window, and display intro message
-- in reverse video.
WRITE_DICT(TPSTATUS, 'DLST', TPTSSP_TITLE, status)
    Chk_Stat( 'TPTSSP_TITLE not written')
-- Write dictionary element,TPTSSP_FK1, from DLST dictionary.
-- Which will display "[TYPE]" to the function line window.
WRITE_DICT(TPFUNC, 'DLST', TPTSSP_FK1, status)
    Chk_Stat( 'TPTSSP_FK1 not written')
-- Write dictionary element, TPTSSP_CLRSC, from DLST dictionary.
-- Which will clear the teach pendant display window.
WRITE_DICT(TPDISPLAY, 'DLST', TPTSSP_CLRSC, status)
    Chk_Stat( 'TPTSSP_CLRSC not written')
-- Write dictionary element, TPTSSP_INSTR, from DLST dictionary.
-- Which will display instructions to the prompt line window.
WRITE_DICT(TPPROMPT, 'DLST', TPTSSP_INSTR, status)
    Chk_Stat( 'TPTSSP_INSTR not written')
-----
----- Section 4-E: Determine the number of menu options
-----
-- Read the dictionary element, TPTSSP_NUM, from DLST
-- dictionary, into the first element of list_data.
-- list_data[1] will be an ASCII representation of the number of
-- menu options. last_line will be returned with the number of
-- lines/elements used in list_data.
READ_DICT('DLST', TPTSSP_NUM, list_data, 1, last_line, status)
    Chk_Stat( 'TPTSSP_NUM not read')

```

```
-- convert the string into the INTEGER, num_options
CNV_STR_INT(list_data[1], num_options)
```

**Figure B.11 (f) Display List from Dictionary File - Write Elements to the Screen**

```
-----
---- Section 4-F: Initialize the data structure, display_data
---- Which is used to display the list of menu
---- options.
-----

-- Initialize the display data structure
-- In this example we only deal with window 1.
display_data.win_start[1] = 1 -- Starting row for window 1.
display_data.win_end[1] = 10 -- Ending row for window 1.
display_data.curr_win = 0 -- The current window to display,
-- where zero (0) specifies first
-- window.
display_data.cursor_row = 1 -- Current row the cursor is on.
display_data.lins_per_pg = 10 -- The number of lines scrolled
-- when the user pushes SHIFT Up
-- or Down. Usually it is the
-- same as the window size.
display_data.curs_st_col[1] = 0 -- starting column for field 1
display_data.curs_en_col[1] = 0 -- ending column for field 1,
-- will be updated a little
-- later
display_data.curr_field = 0 -- Current field, where
-- zero (0) specifies the
-- first field
display_data.last_field = 0 -- Last field in the list
-- (only using one
-- field in this example).
display_data.curr_it_num = 1 -- Current item number the
-- cursor is on.
display_data.sob_it_num = 1 -- Starting item number.
display_data.eob_it_num = num_options -- Ending item number,
-- which is the number
-- of options read in.
display_data.last_it_num = num_options -- Last item number,
-- also the number of
-- options read in
-- Make sure the window end is not beyond total number of elements
-- in list.
IF display_data.win_end[1] > display_data.last_it_num THEN
    display_data.win_end[1] = display_data.last_it_num --reset to
--last item
ENDIF
-- Read dictionary element, TPTSSP_MENU, from dictionary DLST.
-- list_data will be populated with the menu list information
-- list_data[1] will contain the first line of information from
-- the TPTSSP_MENU and list_data[last_line] will contain the last
-- line of information read from the dictionary.
READ_DICT('DLST', TPTSSP_MENU, list_data, 1, last_line, status)
Chk_Stat('Reading menu list failed')
-- Determine longest list element & reset cursor end column for
-- first field.
FOR i = 1 TO last_line DO
    IF (STR_LEN(list_data[i]) > display_data.curs_en_col[1]) THEN
        display_data.curs_en_col[1] = STR_LEN(list_data[i])
```

```

ENDIF
ENDFOR

```

**Figure B.11 (g) Display List from Dictionary File - Initialize Display Data**

```

-----  

---- Section 4-G:    Display the list.  

-----  

-- Initial Display the menu list.  

DISCTRL_LIST(TPDISPLAY, display_data, list_data, DC_DISP, status)  

    Chk_Stat('Error displaying list')  

-- Open a file to the TPDISPLAY window with PASSALL and FIELD  

-- attributes and NOECHO  

SET_FILE_ATR(kb_file, ATR_PASSALL)    -- Get row teach pendant  

                                         -- input  

SET_FILE_ATR(kb_file, ATR_FIELD)      -- so that a single key  

                                         -- will satisfy the reads.  

SET_FILE_ATR(kb_file, ATR_NOECHO)     -- don't echo the keys back  

                                         -- to the screen  

OPEN FILE Kb_file ('RW', 'KB:TPKB')   -- open a file to the Teach  

                                         -- pendant keyboard (keys)  

status = IO_STATUS(Kb_file)  

Chk_Stat('Error opening TPKB')  

act_pending = 0  

done = FALSE  

-----  

---- Section 4-H:    Control cursor movement within the list  

-----  

REPEAT      -- Wait for a key input  

    READ Kb_file (str:::1)  

    key = ORD(str,1)  

    key = key AND 255    -- Convert the key to correct value.  

    SELECT key OF        -- Decide how to handle key inputs  

        CASE (KY_UP_ARW) : -- up arrow key pressed  

            IF act_pending <> 0 THEN  -- If a menu item was selected...  

                -- Clear confirmation prompt  

                WRITE_DICT(TPPROMPT, 'DLST', TPTSSP_CLRSC, status)  

                -- Clear confirmation function keys  

                WRITE_DICT(TPFUNC, 'DLST', TPTSSP_CLRSC, status)  

        ENDIF  

        DISCTRL_LIST(TPDISPLAY, display_data, list_data, DC_UP,  

status)  

        Chk_Stat ('Error displaying list')  

        CASE (KY_DN_ARW) : -- down arrow key pressed  

            IF act_pending <> 0 THEN -- If a menu item was selected...  

                -- Clear confirmation prompt  

                WRITE_DICT(TPPROMPT, 'DLST', TPTSSP_CLRSC, status)  

                -- Clear confirmation function keys  

                WRITE_DICT(TPFUNC, 'DLST', TPTSSP_CLRSC, status)  

        ENDIF  

        DISCTRL_LIST(TPDISPLAY, display_data, list_data,  

DC_DN, status)  

        Chk_Stat ('Error displaying list')  

CASE (KY_ENTER) :  

    -- Perform later  

CASE (KY_F4) : -- "YES" function key pressed  

    IF act_pending <> 0 THEN -- If a menu item was selected...  

        -- Clear confirmation prompt  

        WRITE_DICT(TPPROMPT, 'DLST', TPTSSP_CLRSC, status)  

        -- Clear confirmation function keys

```

```

        WRITE_DICT(TPFUNC, 'DLST', TPTSSP_CLRSC, status)
        IF act_pending = num_options THEN
            -- Exit the routine
            done = TRUE
        ENDIF
        -- Clear action pending
        act_pending = 0
    ENDIF
CASE (KY_F5) : -- "NO" function key pressed
    -- Clear confirmation prompt
    WRITE_DICT(TPPROMPT, 'DLST', TPTSSP_INSTR, status)
    -- Clear confirmation function keys
    WRITE_DICT(TPFUNC, 'DLST', TPTSSP_CLRSC, status)
    -- Clear action pending
    act_pending = 0
ELSE :
    -- User entered an actual item number. Calculate which
    -- row the cursor should be on and redisplay the list.
    IF ((key > 48) AND (key <= (48 + num_options))) THEN
        -- Translate number to a row
        key = key - 48
        display_data.cursor_row = key
        DISCTRL_LIST(TPDISPLAY, display_data, list_data, DC_DISP, status)
        Chk_Stat ('Error displaying list')
        key = KY_ENTER
    ENDIF
ENDSELECT
IF key = KY_ENTER THEN -- User has specified an action
    -- Write confirmation prompt for selected item
    WRITE_DICT (TPPROMPT, 'DLST',
               (TPTSSP_CNF1 - 1 + display_data.cursor_row), status)
    -- Display confirmation function keys
    WRITE_DICT(TPFUNC, 'DLST', TPTSSP_FK2, status)
    -- Set action pending to selected item
    act_pending = display_data.cursor_row -- this is the item selected
ENDIF
UNTIL done -- repeat until the user selects the exit option

```

**Figure B.11 (h) Display List from Dictionary File - Control Cursor Movement**

```

----- Section 4-I: Cleanup before exiting program -----
-- Clear the TP USER menu windows
WRITE_DICT(TPDISPLAY, 'DLST', TPTSSP_CLRSC, status)
WRITE_DICT(TPSTATUS, 'DLST', TPTSSP_CLRSC, status)
-- Close the file connected to the TP keyboard.
CLOSE FILE Kb_file
-- Close the error log file if it is open.
IF opened THEN
    close file Err_File
ENDIF
Write TPPROMPT (cr,'Example Finished ')
REMOVE_DICT ( 'LIST', dp_default, status) -- remove the dictionary
    write ('remove dict', status,cr)
    Chk_Stat ('Removing dictionary')
ACT_SCREEN(old_screen, old_screen, status) -- activate the
                                            -- previous screen
    Chk_stat ('Activating old screen')
DET_WINDOW('ERR', 'LIST', status) -- Detach all the windows
    Chk_stat ('Detaching ERR window') -- that were

```

```

DET_WINDOW('T_ST', 'LIST', status) -- previously attached.
    Chk_stat ('Detaching T_ST window')
DET_WINDOW('T_FU', 'LIST', status)
    Chk_stat ('Detaching T_FU window')
DET_WINDOW('T_PR', 'LIST', status)
    Chk_stat ('Detaching T_PR window')
DET_WINDOW('T_FK', 'LIST', status)
    Chk_stat ('Detaching T_FK window')
END DCLST_EX

```

**Figure B.11 (i) Display List from Dictionary File - Cleanup and Exit Program**

## B.11.1 Dictionary Files

---

This ASCII dictionary file is used to create a teach pendant screen which is used with DCLST\_EX.KL. For more information on DCLST\_EX.KL refer to [Section B.11, DISPLAYING A LIST FROM A DICTIONARY FILE](#).

```

.kl dclist
*
$-,TPTSSP_TITLE &home &reverse "Karel DISCTRL_LIST
Example
"
&standard
$-,TPTSSP_CLRSC &home &clear_2_eow
$-,TPTSSP_FK1 &home" [TYPE] "
$-,TPTSSP_FK2 &home" YES NO "
$-,TPTSSP_INSTR &home "Press 'ENTER' or number key to select."
&clear_2_eol
* Add menu options here, "Exit" must be last option
* TPTSSP_NUM specifies the number of menu options
$-,TPTSSP_NUM "14"
$-,TPTSSP_MENU
" 1 Test Cycle 1" &new_line
" 2 Test Cycle 2" &new_line
" 3 Test Cycle 3" &new_line
" 4 Test Cycle 4" &new_line
" 5 Test Cycle 5" &new_line
" 6 Test Cycle 6" &new_line
" 7 Test Cycle 7" &new_line
" 8 Test Cycle 8" &new_line
" 9 Test Cycle 9" &new_line
" 10 Test Cycle 10" &new_line
" 11 Test Cycle 11" &new_line
" 12 Test Cycle 12" &new_line
" 13 Test Cycle 13" &new_line
" 14 EXIT"
* Confirmations must be in order
$-,TPTSSP_CNF1 &home"Perform test cycle 1? [NO]" &clear_2_eol
$-,TPTSSP_CNF2 &home"Perform test cycle 2? [NO]" &clear_2_eol
$-,TPTSSP_CNF3 &home"Perform test cycle 3? [NO]" &clear_2_eol
$-,TPTSSP_CNF4 &home"Perform test cycle 4? [NO]" &clear_2_eol
$-,TPTSSP_CNF5 &home"Perform test cycle 5? [NO]" &clear_2_eol
$-,TPTSSP_CNF6 &home"Perform test cycle 6? [NO]" &clear_2_eol
$-,TPTSSP_CNF7 &home"Perform test cycle 7? [NO]" &clear_2_eol
$-,TPTSSP_CNF8 &home"Perform test cycle 8? [NO]" &clear_2_eol
$-,TPTSSP_CNF9 &home"Perform test cycle 9? [NO]" &clear_2_eol

```

```
$-,TPTSSP_CNF10 &home"Perform test cycle 10? [NO]" &clear_2_eol
$-,TPTSSP_CNF11 &home"Perform test cycle 11? [NO]" &clear_2_eol
$-,TPTSSP_CNF12 &home"Perform test cycle 12? [NO]" &clear_2_eol
$-,TPTSSP_CNF13 &home"Perform test cycle 13? [NO]" &clear_2_eol
$-,TPTSSP_CNF14 &home"Exit? [NO]" &clear_2_eol
```

**Figure B.11.1 Dictionary File**

## B.12 USING THE DISCTRL\_ALPHA BUILT-IN

This program shows three different ways to use the DISCTRL\_ALPHA built-in. The DISCTRL\_ALPHA built-in displays and controls alphanumeric string entry in a specified window. Refer to [Appendix A, KAREL LANGUAGE ALPHABETICAL DESCRIPTION](#) for more information.

Method 1 allows a program name to be entered using the default value for the dictionary name. See Section 4-A in [Figure B.12 \(c\)](#).

Method 2 allows a comment to be entered using the default value for the dictionary name. See Section 4-B in [Figure B.12 \(c\)](#).

Method 3 uses a user specified dictionary name and element to enter a program name. See Section 4-C in [Figure B.12 \(d\)](#).

This program also posts all errors to the controller.

```
-----
----- DCALP_EX.KL -----
-----
----- Elements of KAREL Language Covered: In Section:
----- Actions:
----- Clauses:
----- Conditions:
----- Data types:
-----          INTEGER           Sec 2
-----          STRING            Sec 2
----- Directives:
-----          COMMENT          Sec 1
-----          INCLUDE          Sec 1
-----          NOLOCKGROUP      Sec 1
----- Built-in Functions & Procedures:
-----          ADD_DICT         Sec 4-C
-----          CHR              Sec 4-A,B,C
-----          DISCTRL_ALPHA     Sec 4-A,B,C
-----          FORCE_SPMENU      Sec 4-A,B,C
-----          POST_ERR          Sec 4-A,B,C
-----          SET_CURSOR        Sec 4-A,B,C
-----          SET_LANG           Sec 4-C
----- Statements:
-----          READ             Sec 4-A,B
-----          WRITE            Sec 4-A,B,C
-----          IF...THEN...ENDIF   Sec 4-A,B,C
----- Reserve Words:
-----          BEGIN            Sec 4
-----          CONST            Sec 2
-----          CR               Sec 4-A,B,C
-----          END              Sec 4-C
```

```

-----      PROGRAM                      Sec 1
-----      VAR                         Sec 2
-----      Predefined File Names:
-----          OUTPUT                     Sec 4-C
-----          TPDISPLAY                  Sec 4-A,B
-----      Predefined Windows:
-----          T_FU                       Sec 4-A,B
-----          C_FU                       Sec 4-C

```

**Figure B.12 (a) Using the DISCTRL\_ALPHA Built-in - Overview**

```

----- Section 1: Program and Environment Declaration
-----
PROGRAM DCALP_EX
%COMMENT      = 'Display Alpha'
%NOLOCKGROUP
%INCLUDE KLEVKEYS    -- Necessary for the KY_ENTER
%INCLUDE DCALPH      -- Necessary for the ALPH_PROG Element, see
                      -- section 4-C
-----
----- Section 2: Constant and Variable Declarations
-----
CONST
  cc_home      = 137
  cc_clear_win = 128
  cc_warn      = 0      -- Value passed to POST_ERR to display
                      -- warning only.
  cc_pause      = 1      -- value passed to POST_ERR to pause program.
VAR
  status       : INTEGER
  device_stat  : INTEGER
  term_char    : INTEGER
  window_name  : STRING[4]
  prog_name    : STRING[12]
  comment      : STRING[40]
-----
----- Section 3: Routine Declaration
-----
```

**Figure B.12 (b) Using the DISCTRL\_ALPHA Built-in - Declaration Section**

```

----- Section 4: Main Program
-----
BEGIN  -- DCALP_EX
-----
----- Section 4-A: Enter a program name from the teach pendant
-----           USER menu
-----
  WRITE (CHR(cc_home), CHR(cc_clear_win)) -- Clear TP USER menu
  FORCE_SPMENU(tp_panel, SPI_TPUSER, 1)   -- Force TP USER menu to
                                             -- be visible
  SET_CURSOR(TPDISPLAY, 12, 1, status)    -- reposition cursor
  WRITE ('prog_name: ')
  prog_name = ''                         -- initialize program name
  DISCTRL_ALPH('t_fu', 12, 12, prog_name, 'PROG', 0, term_char,
  status)
  IF status <> 0 THEN
    POST_ERR(status, '', 0, cc_warn)

```

```

ENDIF
IF term_char = ky_enter THEN -- User pressed the ENTER key
    WRITE (CR, 'prog_name was changed:', prog_name, CR)
ELSE
    WRITE (CR, 'prog_name was not changed')
ENDIF
WRITE (CR, 'Press enter to continue')
READ (CR)
-----
-----
---- Section 4-B: Enter a comment from the teach pendant
-----
-----
WRITE (CHR(cc_home) + CHR(cc_clear_win))-- Clear TP USER menu
SET_CURSOR(TPDISPLAY, 12, 1, status) -- reposition cursor
comment = '' -- Initialize the comment
WRITE ('comment: ') -- Display message
DISCTRL_ALPH('t_fu', 12, 10, comment, 'COMM', 0, term_char, status)
IF status <> 0 THEN -- Verify DISCTRL_ALPH was successful
    POST_ERR(status, '', 0, cc_warn) -- Post the status as a
                                    -- warning
ENDIF
IF term_char = ky_enter THEN
    WRITE (CR, 'comment was changed:', comment, CR) -- Display new
                                                    -- comment
ELSE
    WRITE (CR, 'comment was not changed', CR)
ENDIF
WRITE (CR, 'Press enter to continue')
READ (CR)

```

**Figure B.12 (c) Using the DISCTRL\_ALPHA Built-in - Enter Data from Teach Pendant**

```

-----
---- Section 4-C: This section will perform program name entry from
---- the CRT/KB. The dictionary name and element
---- values are explicitly stated here, instead of
---- using the available default values.
-----
-- Set the dictionary language to English
-- This is useful if you want to use this same code for multiple
-- languages. Then any time you load in a dictionary you check
-- to see what the current language, $language, is and load the
-- correct dictionary.
-- For instance you could have a DCALPHJP.TX file which
-- would be the Japanese dictionary. If the current language,
-- $language, was set to Japanese you would load this
-- dictionary.
SET_LANG ( dp_english, status)
IF (status <> 0) THEN
    POST_ERR (status, '', 0, cc_warn) -- Post the status as a
                                    -- warning
ENDIF
-- Load the dcalpheg.tx file, using ALPH as the dictionary name,
-- to the English language, using DRAM as the memory storage
-- device.
ADD_DICT ('DCALPHEG', 'ALPH', dp_english, dp_dram, status)
IF (status <> 0 ) THEN
    POST_ERR (status, '', 0, cc_pause) -- Post the status and
                                    -- pause the program,

```

```

        -- since the dictionary
        -- must be loaded to continue.
        -- Give control to the CRT/KB
        WRITE OUTPUT (CHR(cc_home) + CHR(cc_clear_win)) --Clear CRT/KB USER
                                                --menu
        FORCE_SPMENU (device_stat, SPI_TPUSER, 1) -- Force the CRT/KB USER
                                                -- menu to be visible
        SET_CURSOR(OUTPUT, 12, 1, status)          -- Reposition the cursor
        WRITE OUTPUT ('prog_name: ')
        prog_name = ''                           -- Initialize program name

DISCTRL_ALPH('c_fu',12,12,prog_name,'ALPH',alph_prog,term_char,status)
)--DISCTRL_ALPHA uses the ALPH dictionary and ALPH_PROG dictionary
--element
    IF status <> 0 THEN                      -- Verify DISCTRL_ALPH was
                                                -- successful.
        POST_ERR(status, '', 0, cc_warn)       -- Post returned status to
                                                -- the error ('err') window.
    ENDIF
    IF term_char = ky_enter THEN
        WRITE (CR, 'prog_name was changed:', prog_name, CR)
    ELSE
        WRITE (CR, 'prog_name was NOT changed.', CR)
    ENDIF
    device_stat = tp_panel                     -- Make sure to reset
END DCALP_EX

```

**Figure B.12 (d) Using the DISCTRL\_ALPHA Built-in - Enter Data from CRT/KB**

## B.12.1 Dictionary Files

This ASCII dictionary file is used to write text to the specified screen. DCALPH\_EG.UTX is also used with DCAL\_EX.KL. For more information on DCAL\_EX.KL, refer to [Section B.12, USING THE DISCTRL\\_ALPHA BUILT-IN](#).

```

.KL DCALPH
$, alpha_prog
" PRG      MAIN      SUB      TEST      >"&new_line
" PROC     JOB       MACRO    TEST4     >"&new_line
" TEST1   TEST2     TEST3    >

```

**Figure B.12.1 DCALPHEG.UTX Dictionary File**

## B.13 APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM

This program copies a teach pendant program and then applies an offset to the positions within the newly created program. This is useful when you create the teach pendant program offline and then realized that all the teach pendant positions are off by some determined amount. However, you should be aware that the utility PROGRAM ADJUST is far more adequate for this job.

```

-----  
---- CPY_TP.KL
-----
```

Elements of KAREL Language Covered:	In Section:
Actions:	
Clauses:	
FROM	Sec 3-F
Conditions:	
Data Types:	
ARRAY OF REAL	Sec 2
ARRAY OF STRING	Sec 2
BOOLEAN	Sec 2
INTEGER	Sec 2; 3-B, C, D, E
JOINTPOS	Sec 2
REAL	Sec 2
STRING	Sec 2
XYZWPR	Sec 2
Directives:	
ENVIRONMENT	Sec 1
Built-in Functions & Procedures:	
AVL_POS_NUM	Sec 3-E
CHR	Sec 3-B, 4
CLOSE_TPE	Sec 3-E
CNV_REL_JPOS	Sec 3-E
CNV_JPOS_REL	Sec 3-E
COPY_TPE	Sec 3-E
GET_JPOS_TPE	Sec 3-E
GET_POS_TYP	Sec 3-E
GET_POS_TPE	Sec 3-E
OPEN_TPE	Sec 3-E
PROG_LIST	Sec 3-B
SELECT_TPE	Sec 3-E
SET_JPOS_TPE	Sec 3-E
SET_POS_TPE	Sec 3-E
Statements:	
FOR...ENDFOR	Sec 3-B, D, E
IF ...THEN...ELSE...ENDIF	Sec 3-A, B, C, E
READ	Sec 3-B, C, D
REPEAT...UNTIL	Sec 3-B, C, D
RETURN	Sec 3-E
ROUTINE	Sec 3-A, B, C, D, E, F
SELECT...ENDSELECT	Sec 3-E
WRITE	Sec 3-B, C, D, E, 4
Reserve Words:	
BEGIN	Sec 3-A, B, C, D, E; 4
CONST	Sec 2
CR	Sec 3-B, C, D, E
END	Sec 3-A, B, C, D, E; 4
PROGRAM	Sec 1
VAR	Sec 2

**Figure B.13 (a) Applying Offsets to Copied Teach Pendant Program - Overview**

----- Section 1: Program and Environment Declaration -----  
PROGRAM CPY\_TP  
%ENVIRONMENT TPE -- necessary for all xxx TPE built-ins

```
%ENVIRONMENT BYNAM -- necessary for PROG_LIST built-in
-----
---- Section 2: Constant and Variable Declaration
-----

CONST
  ER_WARN = 0 -- warning constant for use in POST_ERR
  SUCCESS = 0 -- success constant
  JNT_POS = 9 -- constant for GET_POS_TYP
  XYZ_POS = 2 -- constant for GET_POS_TYP
  MAX_AXS = 9 -- Maximum number of axes JOINTPOS has

VAR
  from_prog: STRING[13] -- TP program name to be copied FROM
  to_prog : STRING[13] -- TP program name to be copied TO
  over_sw : BOOLEAN -- Decides whether to overwrite an existing
                      -- program when performing COPY_TPE
  status   : INTEGER -- Holds error status from the built-in calls
  off_xyz  : XYZWPR -- Offset amount for the XYZWPR positions
  jp_off   : ARRAY [9] of REAL -- Offset amount for the JOINT
                               -- positions
  new_xyz  : XYZWPR -- XYZWPR which has offset applied
  org_xyz  : XYZWPR -- Original XYZWPR from to_prog
  new_jpos : JOINTPOS -- JOINTPOS which has the offset applied
  org_jpos : JOINTPOS -- Original JOINTPOS from to_prog
  open_id  : INTEGER -- Identifier for the opened to_prog
  jp_org   : ARRAY [9] of REAL -- REAL representation of org_jpos
  jp_new   : ARRAY [9] of REAL -- REAL representation of jp_new
```

**Figure B.13 (b) Applying Offsets to Copied Teach Pendant Program - Declaration Section**

```
-----
---- Section 3: Routine Declaration
-----

----- Section 3-A: CHK_STAT Declaration
----- Tests whether the status was successful or
----- not. If the status was not successful the
----- status is posted
-----

ROUTINE chk_stat (rec_stat: integer)
begin
  IF (rec_stat <> SUCCESS) THEN -- if rec_stat is not SUCCESS
    -- then post the error
    POST_ERR (rec_stat, '', 0, ER_WARN) -- Post the error to the
                                         -- system.
  ENDIF
END chk_stat
-----

----- Section 3-B: GetFromPrg Declaration
----- Generate a list of loaded TPE programs.
----- Lets the user select one of these programs
----- to be the program to be copied, ie FROM_prog
-----

ROUTINE GetFromPrg
VAR
  tp_type   : INTEGER -- Types of program to list
  n_skip    : INTEGER -- Index into the list of programs
  format    : INTEGER -- What type of format to store programs in
  n_progs   : INTEGER -- Number of programs returned in prg_name
  prg_name  : ARRAY [8] of STRING[20] --Program names returned from
                                         --PROG_LIST
```

```

status      : INTEGER -- Status of PROG_LIST call
f_index     : INTEGER -- Fast index for generating the program
-- listing.
arr_size    : INTEGER -- Array size of prg_name
prg_select : INTEGER -- Users selection for which program to copy
indx       : INTEGER -- FOR loop counter which displays prg_name

```

**Figure B.13 (c) Applying Offsets to Copied Teach Pendant Program - Declare Routines**

```

BEGIN
  f_index = 0      -- Initialize the f_index
  n_skip   = 0      -- Initialize the n_skip
  tp_type  = 2      -- find any TPE program
  format   = 1      -- return just the program name in prg_name
  n_progs = 0      -- Initialize the n_progs
  arr_size = 8      -- Set equal to the declared array size of prg_name
  prg_select = 0    -- Initialize the program selector
REPEAT
  WRITE (chr(128),chr(137)) -- Clear the TP USER screen
  -- Get a listing of all TP program which begin with "TEST"

PROG_LIST('TEST*',tp_type,n_skip,format,prg_name,n_progs,status,f_index)
  chk_stat (status) --Check status from PROG_LIST
  FOR indx = 1 to n_progs DO
    WRITE (indx,':',prg_name[indx], CR) -- Write the list of
                                         -- programs out
  ENDFOR
  IF (n_skip > 0) OR ( n_prog >0) THEN
    WRITE ('select program to be copied:',CR)
    WRITE ('PRESS -1 to get next page of programs:')
    REPEAT
      READ (prg_select) -- get program selection
      UNTIL ((prg_select >= -1) AND (prg_select <= n_progs)
              AND (prg_select <> 0))
  ELSE
    WRITE ('no TP programs to COPY', CR)
    WRITE ('Aborting program, since need',CR)
    WRITE ('at least one TP program to copy.',CR)
    ABORT
  ENDIF
  -- Check if listing is complete and user has not made a
  -- selection.
  IF ((prg_select = -1) AND (n_progs < arr_size)) THEN
    f_index = 0      --reset f_index to re-generate list.
    n_progs = arr_size --set so the REPEAT/UNTIL will continue
  ENDIF
  -- Check if user user has made a selection
  IF (prg_select <> -1) then
    from_prog = prg_name[prg_select] -- Set from_prog to name
                                      -- selected
    n_progs = 0          -- Set n_prog to stop looping.
  ENDIF
  UNTIL (n_progs < arr_size)
END GetFromPrg

```

**Figure B.13 (d) Applying Offsets to Copied Teach Pendant Program - Generate Program List for User**

-----  
---- Section 3-C: GetOvrSw Declaration

```
---- Ask user whether to overwrite the copied
---- program, TO_prog, if it exists.
-----
ROUTINE GetOvrSw
VAR
    yesno : INTEGER
BEGIN
    WRITE (CR, 'If Program already exists do you want',CR)
    WRITE ('to overwrite the file Yes:1, No:0 ? ')
    REPEAT
        READ (yesno)
    UNTIL ((yesno = 0) OR( yesno = 1))
    IF yesno = 1 then --Set over_sw so program is overwritten if it
                      --exists
        over_sw = TRUE
    ELSE           --Set over_sw so program is NOT overwritten if
                  --it exists
        over_sw = FALSE
    ENDIF
END GetOvrSw
```

**Figure B.13 (e) Applying Offsets to Copied Teach Pendant Program - Overwrite or Delete Program**

```
----- Section 3-D: GetOffset Declaration
----- Have the user input the offset for both
----- XYZWPR and JOINTPOS positions.
-----
ROUTINE GetOffset
VAR
    yesno : INTEGER
    index : INTEGER
BEGIN
    --Get the XYZWPR offset, off_xyz
    REPEAT
        WRITE ( 'Enter offset for XYZWPR positions',CR)
        WRITE (' X = ')
        READ (off_xyz.x)
        WRITE (' Y = ')
        READ (off_xyz.y)
        WRITE (' Z = ')
        READ (off_xyz.z)
        WRITE (' W = ')
        READ (off_xyz.w)
        WRITE (' P = ')
        READ (off_xyz.p)
        WRITE (' R = ')
        READ (off_xyz.r)
        --Display the offset values the user input
        WRITE (' Offset XYZWPR position is',CR, off_xyz,CR)
        WRITE ('Is this offset correct? Yes:1, No:0 ? ')
        READ (yesno)
    UNTIL (yesno = 1)      -- enter offset amounts until the user
                           -- is satisfied.
    --Get the JOINTPOS offset, jp_off
    REPEAT
        WRITE ( 'Enter offset for JOINT positions',CR)
        FOR indx = 1 TO 6 DO      -- loop for number of robot axes
            WRITE (' J',indx,' = ')
            READ (jp_off[indx])
```

```

ENDFOR  WRITE ('JOINT position offset is', CR)
FOR indx = 1 TO 6 DO
    write ( jp_off[indx],CR) -- Display the values the user
                               -- input
ENDFOR
WRITE ('Is this offset correct? Yes:1, No:0 ? ')
READ  (yesno)
UNTIL (yesno = 1)      -- Enter offset amounts until the user
                        -- is satisfied
END GetOffset

```

**Figure B.13 (f) Applying Offsets to Copied Teach Pendant Program - Input Offset Positions**

```

-----  

---- Section 3-E: ModifyPrg Declaration  

---- Apply the offsets to each position within the  

---- TP program  

-----  

ROUTINE ModifyPrg
VAR
    pos_typ : INTEGER -- The type of position returned from GET_POS_TYP
    num_axs : INTEGER -- The number of axes if position is a JOINTPOS
                       -- type
    indx_pos: INTEGER -- FOR loop counter, that increments through TP
                       -- position
    group_no: INTEGER -- The group number of the current position
                       -- setting.
    num_pos : INTEGER -- The next available position number within TP
                       -- program
    indx_axs: INTEGER -- FOR loop counter, increments through REAL
                       -- array
BEGIN
    SELECT_TPE ('', status) -- Make sure the to_prog is currently not
                           -- selected
    to_prog = 'MDFY_TP' -- Set the to_prog to desired name.
    ----- Copy the from_prog to to_prog -----
    COPY_TPE (from_prog, to_prog, over_sw, status)
    chk_stat(status) -- check status of COPY_TPE
    --- If the user specified not to overwrite the TPE program and
    --- the status returned is 7015, "program already exist",
    --- then quit the program. This will mean not altering the already
    --- existing to_prog.
    IF ((over_sw = FALSE) AND (status = 7015)) THEN
        WRITE ('ABORTING:: PROGAM ALREADY EXISTS!',CR)
        RETURN
    ENDIF
    --- Open the to_prog with the Read/Write access
    OPEN_TPE (to_prog, TPE_RWACC, TPE_RDREJ, open_id, status)
    chk_stat(status) -- check status of OPEN_TPE
    group_no = 1
    --- apply offset to each position within to_prog
    --- The current number of position that the TPE program has is
    --- num_pos -1
    FOR indx_pos = 1 to num_pos-1 DO
        -- Get the DATA TYPE of each position within the to_prog
        -- If it is a JOINTPOS also get the number of axes.
        GET_POS_TYP (open_id, indx_pos, group_no, pos_typ, num_axs,
                     status)
        chk_stat (status)
        WRITE('get_pos_typ status', status,cr)
        -- Decide if the position, indx_pos, is a JOINTPOS or a XYZWPR

```

```

SELECT pos_typ OF
CASE (JNT_POS):
    -- The position is a JOINTPOS
    FOR indx_axs = 1 TO MAX_AXS DO  -- initialize with default
        -- values
        jp_org[indx_axs] = 0.0  -- This avoids problems with the
        jp_new[indx_axs] = 0.0  -- CNV_REL_JPOS
    ENDFOR
    -- get the JOINTPOS P[indx_pos] from to_prog -----
    org_jpos = GET_JPOS_TPE (open_id, indx_pos, status)
    chk_stat (status)
    -- Convert the JOINTPOS to a REAL array, in order to perform
    -- offset
    CNV_JPOS_REL (org_jpos, jp_org, status)
    chk_stat (status)
    -- Apply the offset to the REAL array
    FOR indx_axs = 1 to num_axs DO
        jp_new[indx_axs] = jp_org[indx_axs] + jp_off[indx_axs]
    ENDFOR
    -- Converted back to a JOINTPOS.
    -- The input array, jp_new, must not have any uninitialized
    -- values or the error 12311 - "Data uninitialized"
    -- will be posted.
    -- This is why we previously set all the values to zero.
    CNV_REL_JPOS (jp_new, new_jpos, status)
    chk_stat (status)
    -- Set the new offset position, new_jpos, into the indx_pos
    SET_JPOS_TPE (open_id, indx_pos, new_jpos, status)
    chk_stat (status)
    write ('indx_pos', indx_pos, 'new_jpos', cr, new_jpos, cr)
CASE (XYZ_POS): -- The position is a XYZWPR
    -- Get the XYZWPR position P[indx_pos] from to_prog
    org_xyz = GET_POS_TPE (open_id, indx_pos, status)
    chk_stat (status) -- Check status from GET_POS_TPE

```

**Figure B.13 (g) Applying Offsets to Copied Teach Pendant Program - Apply Offsets to Positions**

```

-- Apply offset to the XYZWPR
new_xyz.x = org_xyz.x + off_xyz.x
new_xyz.y = org_xyz.y + off_xyz.y
new_xyz.z = org_xyz.z + off_xyz.z
new_xyz.w = org_xyz.w + off_xyz.w
new_xyz.p = org_xyz.p + off_xyz.p
new_xyz.r = org_xyz.r + off_xyz.r
    --Set the new offset position, new_xyz, into the indx_pos
    SET_POS_TPE (open_id, indx_pos, new_xyz, status)
    chk_stat (status) -- Check status from SET_POS_TPE
ENDSELECT
ENDFOR
---Close TP program before quitting program
CLOSE_TPE (open_id, status)
chk_stat (status) --Check status from CLOSE_TPE
END ModifyPrg
-----
----- Section 3-F: TP_CLS Declaration
----- Clears the TP USER Menu screen and forces it to
----- become visible. The actual code resides in
----- ROUT_EX.KL
-----
ROUTINE TP_CLS FROM rout_ex
-----
```

```

----- Section 4: Main Program
-----
BEGIN -- CPY_TP
  tp_cls                      -- Clear the TP USER Menu screen
  GetFromPrg                   -- Get the TPE program to copy FROM
  GetOvrSw                     -- Get the TPE program name to copy TO
  GetOffset                    -- Get the offset for modifying
  ModifyPrg                   -- Modify the copied program by the offset
END CPY_TP

```

**Figure B.13 (h) Applying Offsets to Copied Teach Pendant Program - Clears TP User Menu**

## B.14 iVISION CUSTOM SCREEN USING V\_CSAPI BUILT-INS

---

The sample program of the custom screen consists of the client-side program CUSTOM.HTM, server-side program VSBLTIN.KL, vision process VP2S, vision overrides VAL\_SCORE, VAL\_CONTRAST. This section introduces the client-side program CUSTOM.HTM and the server-side program V\_CSAPI\_\* built-ins. Create the vision program VP2S, vision overrides VAL\_SCORE, VAL\_CONTRAST in advance.

- Create vision process VP2S. This vision process should have a GPM Locator tool.
- The vision process VP2S and the vision tool GPM Locator tool 1 should be selected in the both vision overrides VAL\_SCORE and VAL\_CONTRAST.
- As for VAL\_SCORE, Score Threshold should be selected as a parameter. As for VAL\_CONTRAST, Contrast Threshold should be selected as a parameter.

Figure B.14 shows a client-side sample of a custom iVision screen CUSTOM.HTM defining the appearance and behavior of the screen. This file sends a request to the sample program VSBLTIN.KL on the server side and processes the returned JSON format response.

For details, please refer to [Figure B.14](#) below. Also, please refer to the related contents in the *Internet Options Setup and Operations Manual (MAROUIN901017IE)* or the *Ethernet Function OPERATOR'S MANUAL (B-82974EN)*.

```

<!DOCTYPE html>
<!--
-----
custom.htm - Custom page
-----
-->
<html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=11;IE=7">
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
<title>Custom</title>
</head>
<style type="text/css">
tr {
height: 40px;
vertical-align: top;
}
button {
margin: 10px 20px 0 0;

```

```

}
.text {
width: 100px;
}
</style>
<body tabindex="-1">
<table>
<colgroup>
<col>
<col>
</colgroup>
<tr>
<td>
<label id="lblScore" class="text">Score Threshold</label>
</td>
<td>
<!-- Text box for entering the score threshold -->
<input type="text" id="txtScore" size="10"
onchange="callKarel(I_FUNC_SVAL)"></input>
</td>
</tr>
<tr class="row-item">
<td>
<label id="lblContrast" class="text">Contrast Threshold</label>
</td>
<td>
<!-- Text box for entering the contrast threshold -->
<input type="text" id="txtContrast" size="10"
onchange="callKarel(I_FUNC_SVAL)"></input>
</td>
</tr>
</table>
<!-- Button for test run -->
<button id="btnSnap" type="button" class="text"
onclick="callKarel(I_FUNC_TSTR)">
SNAP+FIND
</button>
<!-- Button to undo the changed parameters -->
<button id="btnReset" type="button" class="text"
onclick="callKarel(I_FUNC_RSDT)">
RESET
</button>
<!-- Button to save the vision process -->
<button id="btnSave" type="button" class="text"
onclick="callKarel(I_FUNC_SVDT)">
SAVE
</button>
<script type="text/javascript">
var I_FUNC_GVAL = 1;
var I_FUNC_SVAL = 2;
var I_FUNC_TSTR = 3;
var I_FUNC_SVDT = 4;
var I_FUNC_RSDT = 5;
var I_FUNC_NMST = 6;
var READYSTATE_COMPLETE = 4;
var RESPONSE_OK = 200;
var RESPONSE_MULTIPLE_CHOISE = 300;
var RESPONSE_NOT_MODIFIED = 304;
var RESPONSE_TP_OK = 1223;
window.onload = function() {
callKarel(I_FUNC_GVAL);

```

```

};

function callKarel(funcMode) { // Function that sends a request to
VSBLTIN
var prgReq = new XMLHttpRequest();
var url = "/KARELCMD/VSBLTIN?s_func_mode=" + funcMode; // URL of
request to VSBLTIN
switch(funcMode) {
case I_FUNC_GVAL:
case I_FUNC_TSTR:
case I_FUNC_NMST:
case I_FUNC_SVDT:
case I_FUNC_RSDT:
/* nop */
break;
case I_FUNC_SVAL:
url += "&s_score=" + document.getElementById("txtScore").value +
"&s_contrast=" + document.getElementById("txtContrast").value;
break;
default:
break;
}
prgReq.onreadystatechange = chkReadyStateChgPrg;
prgReq.open("GET", url, true);
prgReq.send();
function chkReadyStateChgPrg() {
if (prgReq.readyState === READYSTATE_COMPLETE) {
if (((prgReq.status >= RESPONSE_OK) &&
(prgReq.status < RESPONSE_MULTIPLE_CHOISE)) ||
(prgReq.status === RESPONSE_NOT_MODIFIED) ||
(prgReq.status === RESPONSE_TP_OK)) {
getReqData(prgReq);
}
prgReq.abort();
}
}
}

function getReqData(request) { // Function that processes the
response from VSBLTIN
var parseData;
if (window.JSON === undefined) {
parseData = eval("(" + request.responseText + ")");
}
else {
parseData = JSON.parse(request.responseText);
}
var rCode = parseData.status;
if (rCode !== 0) {
alert("Err Status:" + rCode);
}
else {
switch(parseData.funcMode) {
case I_FUNC_GVAL:
document.getElementById("txtScore").value = parseData.visionData[0];
document.getElementById("txtContrast").value =
parseData.visionData[1];
callKarel(I_FUNC_NMST);
break;
case I_FUNC_SVAL:
case I_FUNC_TSTR:
case I_FUNC_RSDT:
case I_FUNC_SVDT:
}
}
}
}

```

```
callKarel(I_FUNC_GVAL);
break;
case I_FUNC_NMST:
var disabled = parseData.visionData[0] < 1;
document.getElementById("btnReset").disabled = disabled;
document.getElementById("btnSave").disabled = disabled;
break;
default:
break;
}
}
}
</script>
</body>
</html>
```

**Figure B.14 Customized Screen Sample**

EFFMAN  
QUIRIONR

# C KCL COMMAND ALPHABETICAL DESCRIPTION

---



---

This section describes each KCL command in alphabetical order. Each description includes the purpose of the command, its syntax, and details of how to use it. Examples of each command are also provided.

The following notation is used to describe KCL command syntax:

- <> indicates optional arguments to a command
- | indicates a choice which must be made
- { } indicates an item can be repeated
- file\_spec: <device\_name:><\host\_name><path\_name><file\_name>.file\_type |  
  <device\_name:><\host\_name>'host\_specific\_name'
- path\_name: <file\_name><dir\dir\...>
- file\_name: maximum of 36 characters, no file type

device\_name: is a two to five-character optional field, followed by a colon. The first character is a letter, the remaining characters must be alphanumeric. If this field is left blank, the default device from the system variable \$DEVICE will be used.

host\_name: is a one to eight character optional field. The host\_name selects the network node that receives this command. It must be preceded by two backslashes and separated from the remaining fields by a backslash.

path\_name : file\_name\<path\_name> - is a recursively defined optional field, each field consisting of a maximum of 36 characters. It is used to select the file subdirectory. The root or source directory is handled as a special case. It is designated by a file\_name of zero length. For example, access to the subdirectory SYS linked off of the root would have path name '\SYS'. A fully qualified file\_spec using this path\_name would look like this, 'C1:\HOST\SYS\FILE.KL'.

file\_name: from one to 36 characters

file\_type: from zero to three characters

KCL commands can be abbreviated allowing you to type in fewer letters as long as the abbreviated version remains unique among all keywords. For example, ABORT can be AB but CONTINUE must be CONT to distinguish it from CONDITION.

path\_names, file\_names, and file\_types that contain special characters or begin with numbers can be specified as a host\_specific\_name inside single quotes. FR:\'00\' is valid, FR:\'00\test.kl' is valid, FR:\'00'\test.kl is invalid because the host\_specific\_name must be last.

KCL commands that have <prog\_name> as part of the command syntax will use the default program if none is specified. KCL commands that have <file\_name> as part of the command syntax will use the default program as the file name if none is specified.

## C.1 ABORT command

---

**Syntax:** ABORT <( prog\_name ) | ALL > <FORCE>

where:

prog\_name : the name of any KAREL or TP program which is a task

ALL : aborts all running or paused tasks

FORCE : aborts the task even if the NOABORT attribute is set. FORCE only works with ABORT prog\_name; FORCE does not work with ABORT ALL

**Purpose:** Aborts the specified running or paused task. If prog\_name is not specified, the default program is used.

Execution of the current program statement is completed before the task aborts except for the current motion, DELAY, WAIT, or READ statements, which are canceled.

**Examples:**

```
KCL> ABORT test_prog FORCE
```

```
KCL> ABORT ALL
```

## C.2 APPEND FILE command

---

**Syntax:** APPEND FILE input\_file\_spec TO output\_file\_spec

where:

input\_file\_spec : a valid file specification

output\_file\_spec : a valid file specification

**Purpose:** Appends the contents of the specified input file to the end of the specified output file. The input\_file\_spec and the output\_file\_spec must include both the file name and the file type.

**Examples:**

```
KCL> APPEND FILE flpy:test.kl TO productn.kl
```

```
KCL> APPEND FILE test.kl TO productn.kl
```

## C.3 APPEND NODE command

---

**Syntax:** APPEND NODE <[ prog\_name ]> var\_name

where:

prog\_name : the name of any KAREL or TP program

var\_name : the name of any variable of type PATH

**Purpose:** Appends one node to the end of the specified PATH variable previously loaded in RAM. The appended node value is uninitialized and the index number is one more than the last node index. Execute the KCL> SAVE VARS command to make the change permanent.

**Examples:**

```
KCL> APPEND NODE [test_prog]weld_pth
```

```
KCL> APPEND NODE weld_pth
```

## C.4 CHDIR command

---

**Syntax:** CHDIR <device\_name>\<path\_name>\ or CD <device\_name>\<path\_name>

where:

device\_name : a specified device

path\_name : a subdirectory previously created on the memory card device using the mkdir command. When the chdir command is used to change to a subdirectory, the entire path will be displayed on the teach pendant screen as mc:\new\_dir\new\_file.

The double dot (..) can be used to represent the directory one level above the current directory.

**Purpose:** Changes the default device. If a device\_name is not specified, displays the default device.

**Examples:**

```
KCL> CHDIR rd:\
```

```
KCL> CD
```

```
KCL> CD mc:\a
```

```
KCL> CD ..
```

## C.5 CLEAR ALL command

---

**Syntax:** CLEAR ALL <YES>

where:

YES : confirmation is not prompted

**Purpose:** Clears all KAREL and teach pendant programs and variable data from memory. All cleared programs and variables (if they were saved with the KCL> SAVE VARS command) can be reloaded into memory using the KCL> LOAD command.

**Examples:**

```
KCL> CLEAR ALL
Are you sure? YES
```

```
KCL> CLEAR ALL Y
```

## C.6 CLEAR BREAK CONDITION command

**Syntax:** CLEAR BREAK CONDITION <prog\_name> ( brk\_pnt\_no | ALL)

where:

prog\_name : the name of any KAREL program in memory

brk\_pnt\_no : a particular condition break point

ALL : clears all condition break points

**Purpose:** Clears specified condition break point(s) from the specified or default program.

A condition break point only affects the program in which it is set.

**Examples:**

```
KCL> CLEAR BREAK CONDITION test_prog 3
```

```
KCL> CLEAR BREAK COND ALL
```

## C.7 CLEAR BREAK PROGRAM command

**Syntax:** CLEAR BREAK PROGRAM <prog\_name> ( brk\_pnt\_no | ALL)

where:

prog\_name : the name of any KAREL program in memory

brk\_pnt\_no : a particular program break point

ALL : clears all break points

**Purpose:** Clears specified break point(s) from the specified or default program.

A break point only affects the program in which it is set.

**Examples:**

```
KCL> CLEAR BREAK PROGRAM test_prog 3
```

```
KCL> CLEAR BREAK PROG ALL
```

## C.8 CLEAR DICT command

**Syntax:** CLEAR DICT dict\_name <( lang\_name | ALL)>

where:

dict\_name : the name of any dictionary to be cleared

lang\_name : the name of the language. The available choices are ENGLISH, JAPANESE, FRENCH, GERMAN, SPANISH or DEFAULT.

ALL : clears the dictionary from all languages

**Purpose:** Clears a dictionary from the specified language or from all languages. If no language is specified, it is cleared from the DEFAULT language only.

**Examples:**

```
KCL> CLEAR DICT tpsy ENGLISH
```

```
KCL> CLEAR DICT tpsy
```

## C.9    CLEAR PROGRAM command

---

**Syntax:** CLEAR PROGRAM <prog\_name> <YES>

where:

prog\_name : the name of any KAREL or teach pendant program in memory

YES : confirmation is not prompted

**Purpose:** Clears the program data from memory for the specified or default program.

**Examples:**

```
KCL> CLEAR PROGRAM test_prog  
Are you sure? YES
```

```
KCL> CLEAR PROG test_prog Y
```

## C.10    CLEAR VARS command

---

**Syntax:** CLEAR VARS <prog\_name> <YES>

where:

prog\_name : the name of any KAREL or teach pendant program with variables

YES : confirmation is not prompted

**Purpose:** Clears the variable and type data associated with the specified or default program from memory.

Variables and types that are referenced by a loaded program are not cleared.

**Examples:**

```
KCL> CLEAR VARS test_prog  
Are you sure? YES
```

```
KCL> CLEAR VARS test_prog Y
```

## C.11 COMPRESS DICT command

---

**Syntax:** COMPRESS DICT file\_name

where:

file\_name : the file name of the user dictionary you want to compress.

**Purpose:** Compresses a dictionary file from the default storage device, using the specified dictionary name. The file type of the user dictionary must be .UTX. The compressed dictionary file will have the same file name as the user dictionary, and be of type .TX.

**See Also:** [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:**

```
KCL> COMPRESS DICT tphcmneg
```

## C.12 COMPRESS FORM command

---

**Syntax:** COMPRESS FORM <file\_name>

where:

file\_name : the file name of the form you want to compress.

**Purpose:** Compresses a form file from the default storage device using the specified form name. The file type of the form must be .FTX. A compressed dictionary file and variable file will be created. The compressed dictionary file will have the same file name as the form file and be of type .TX. The variable file will have a four character file name, that is extracted from the form file name, and be of type .VR. If no form file name is specified, the name FORM is used.

**See Also:** [Chapter 11, DICTIONARIES AND FORMS](#)

**Examples:**

```
KCL> COMPRESS FORM
```

```
KCL> COMPRESS FORM mnpalteg
```

## C.13 CONTINUE command

---

**Syntax:** CONTINUE <( prog\_name ) | ALL>

where:

prog\_name : the name of any KAREL or teach pendant program which is a task

ALL : continues all paused tasks

**Purpose:** Continues program execution of the specified task that has been paused by a hold, pause, or test run operation. If the program is aborted, the program execution is started at the first executable line.

When a task is paused, the **CYCLE START** button on the operator panel has the same effect as the **KCL> CONTINUE** command.

**CONTINUE** is a motion command; therefore, the device from which it is issued must have motion control.

#### Examples:

```
KCL> CONTINUE test_prog
```

```
KCL> CONT ALL
```

## C.14 COPY FILE command

**Syntax:** **COPY <FILE> from\_file\_spec TO to\_file\_spec <OVERWRITE>**

where:

**from\_file\_spec** : a valid file specification

**to\_file\_spec** : a valid file specification

**OVERWRITE** : specifies copy over (overwrite) an existing file

**Purpose:** Copies the contents of one file to another with overwrite option. Allows file transfers between different devices and between the controller and a host system.

The wildcard character (\*) can be used to replace **from\_file\_spec**'s entire file name, the first part of the file name, the last part of the file name, or both the first and last parts of the file name. The file type can also use the wildcard in the same manner. The wildcard character in the **to\_file\_spec** can only replace the entire file name or the entire file type.

#### Examples:

```
KCL> COPY flpy:\test.kl TO rdu:newtest.kl
```

```
KCL> COPY mc:\test_dir\test.kl TO mc:\test_dir\newtest.kl
```

```
KCL> COPY FILE flpy:\*.kl TO rd:*.kl
```

```
KCL> COPY *.* TO fr:
```

```
KCL> COPY FILE *.kl TO rd:\*.bak OVERWRITE
```

```
KCL> COPY FILE flpy:\*main*.kl TO rd:*.OV
```

```
KCL> COPY mdb:*.tp TO mc:
```

## C.15 CREATE VARIABLE command

**Syntax:** **CREATE VARIABLE <[ prog\_name ]> var\_name <IN (CMOS | DRAM) > : data\_type**

where:

prog\_name : the name of any KAREL or TP program

var\_name :data\_type : a valid variable name and data type

**Purpose:** Allows you to declare a variable that will be associated with the specified or default program. You must specify a valid identifier for the var\_name and a valid data\_type.

Only one variable can be declared with the CREATE VAR command. You must enter the KCL> SAVE VARS command to save the declared variable with the program variable data. Use the KCL> SET VARIABLE command to assign a value to a variable.

The following data types are valid (user types are also supported): ARRAY OF BYTE, JOINTPOS, ARRAY OF SHORT, JOINTPOS1 to JOINTPOS9, BOOLEAN, POSITION, REAL, CONFIG, VECTOR, FILE, XYZWPR, XYZWPREXT, INTEGER

You can create multi-dimensional arrays of the above type. A maximum of 3 dimensions may be specified. Paths may only be created from a user defined type.

By default, the variable will be created in temporary memory (in DRAM), and must be recreated every power up. The value will always be reset to uninitialized.

If IN CMOS is specified the variable will be created in permanent memory. The variable's value will be recovered every time the controller is turned on.

**See Also:** [Section C.69, SET VARIABLE command](#)

**Examples:**

```
KCL> CREATE VAR [test_prog]count IN CMOS: INTEGER
```

```
KCL> CREATE VAR vec:ARRAY[3,2,4] OF VECTOR
```

## C.16 DELETE FILE command

**Syntax:** DELETE FILE file\_spec <YES>

where:

file\_spec : a valid file specification

YES : confirmation is not prompted

**Purpose:** Deletes the specified file from the specified storage device. The wildcard character (\*) can be used to replace the entire file name, the first part of the file name, the last part of the file name, or both the first and last parts of the file name. The file type can also use the wildcard in the same manner.

**Examples:**

```
KCL> DELETE FILE testprog.pc  
Are you sure? YES
```

```
KCL> DELETE FILE rd:\testprog.pc YES
```

```
KCL> DELETE FILE rd:\*.* Y
```

## C.17 DELETE NODE command

---

**Syntax:** DELETE NODE <[ prog\_name ]> var\_name [node\_index]

where:

prog\_name : the name of any KAREL or TP program

var\_name : the name of any variable of type PATH

[node\_index] : a node in the path

**Purpose:** Deletes the specified node from the specified PATH variable. The PATH variable must be loaded in memory. Enter the KCL> SAVE VARS command to make the change permanent.

**Examples:**

```
KCL> DELETE NODE [test_prog]weld_pth[4]
```

```
KCL> DELETE NODE weld_pth[3]
```

## C.18 DELETE VARIABLE command

---

**Syntax:** DELETE VARIABLE <[ prog\_name ]> var\_name

where:

prog\_name : the name of any KAREL or TP program with variables

var\_name : the name of any program variable

**Purpose:** Deletes the specified variable from memory. A variable that is linked with loaded p-code cannot be deleted. Enter the KCL> SAVE VARS command to make the change permanent.

**Examples:**

```
KCL> DELETE VARIABLE [test_prog]weld_pth
```

```
KCL> DELETE VAR weld_pth
```

## C.19 DIRECTORY command

---

**Syntax:** DIRECTORY <file\_spec>

where:

file\_spec : a valid file specification

**Purpose:** Displays a list of the files that are on a storage device. If file\_spec is not specified, directory information is displayed for all of the files stored on a specified device. The directory information displayed includes the following:

- The volume name of the device (if specified when the device was initialized)
- The name of the subdirectory, if available

- The names and types of files currently stored on the device and the sizes of the files in bytes
- The number of files, the number of bytes left, and the number of bytes total, if available

The wildcard character (\*) can be used to replace the entire file name, the first part of the file name, the last part of the file name, or both the first and last parts of the file name. The file type can also use the wildcard in the same manner.

**Examples:**

```
KCL> DIRECTORY rd:
```

```
KCL> DIR *.kl
```

```
KCL> DIR *SPOT*.kl
```

```
KCL> CD MC: \test_dir
```

Use the CD command to change to the KCL> DIR subdirectory before you use the DIR command or KCL> DIR \test\_dir\\*.\* to display the subdirectory contents without using the CD command.

## C.20 DISABLE BREAK PROGRAM command

**Syntax:** DISABLE BREAK PROGRAM <prog\_name> brk\_pnt\_no

where:

prog\_name : the name of any KAREL or TP program in memory

brk\_pnt\_no : a particular program break point

**Purpose:** Disables the specified break point in the specified or default program.

**Examples:**

```
KCL> DISABLE BREAK PROGRAM test_prog 3
```

```
KCL> DISABLE BREAK PROG 3
```

## C.21 DISABLE CONDITION command

**Syntax:** DISABLE CONDITION <prog\_name> condition\_no

where:

prog\_name : the name of any KAREL program in memory

condition\_no : a particular condition

**Purpose:** Disables the specified condition in the specified or default program.

**Examples:**

```
KCL> DISABLE CONDITION test_prog
```

```
KCL> DISABLE COND 3
```

## C.22 DISMOUNT command

---

**Syntax:** DISMOUNT device\_name:

where:

device\_name : device to be dismounted

**Purpose:** Dismounts a mounted storage device and indicates to the controller that a storage device is no longer available for reading or writing data.

**Example:**

```
KCL> DISMOUNT rd:
```

## C.23 EDIT command

---

**Syntax:** EDIT <file\_spec>

where:

file\_spec : a valid file specification

**Purpose:** Provides an ASCII text editor which can be used for editing dictionary files, command files and KAREL source files.

If file\_spec is not specified, the default program name is used as the file name and the default file type is .KL (KAREL source code).

If a previous editing session exists, then file\_spec is ignored and the editing session is resumed.

**Examples:**

```
KCL> EDIT startup.cf
```

```
KCL> ED
```

## C.24 ENABLE BREAK PROGRAM

---

**Syntax:** ENABLE BREAK PROGRAM <prog\_name> brk\_pnt\_no

where:

prog\_name : the name of any KAREL or TP program in memory

brk\_pnt\_no : a particular program break point

**Purpose:** Enables the specified break point in the specified or default program.

**Examples:**

```
KCL> ENABLE BREAK PROGRAM test_prog 3
```

```
KCL> ENABLE BREAK PROG 3
```

## C.25 ENABLE CONDITION command

**Syntax:** ENABLE CONDITION <prog\_name> condition\_no

where:

prog\_name : the name of any KAREL program in memory

condition\_no : a particular condition

**Purpose:** Enables the specified condition in the specified or default program.

**Examples:**

```
KCL> ENABLE CONDITION test_prog
```

```
KCL> ENABLE COND 3
```

## C.26 FORMAT command

**Syntax:** FORMAT device\_name : <volume\_name> <YES>

where:

device\_name : the specified device to be initialized

volume\_name : label for the device

YES : confirmation is not prompted

**Purpose:** Formats a specified device. A device must be formatted before storing files on it.

**Examples:**

```
KCL> FORMAT rd:  
Are you sure? YES
```

```
KCL> FORMAT rd: Y
```

## C.27 HELP command

**Syntax:** HELP <command\_name>

where:

command\_name : a KCL command

**Purpose:** Displays on-line help for KCL commands. If you specify a command\_name argument, the required syntax and a brief description of the specified command is displayed.

**Examples:**

```
KCL> HELP LOAD PROG
```

```
KCL> HELP
```

## C.28 HOLD command

---

**Syntax:** HOLD <( prog\_name | ALL )>

where:

prog\_name : the name of any KAREL or TP program

ALL : holds all executing programs

**Purpose:** Pauses the specified or default program that is being executed and holds motion at the current position (after a normal deceleration).

Use the KCL> CONTINUE command or the CYCLE START button on the operator panel to resume program execution.

**Examples:**

```
KCL> HOLD test_prog
```

```
KCL> HO ALL
```

## C.29 INSERT NODE command

---

**Syntax:** INSERT NODE <[ prog\_name ]> var\_name [node\_index]

where:

prog\_name : the name of any KAREL or TP program

var\_name : the name of any variable of type PATH

[node\_index] : a node in the path

**Purpose:** Inserts a node in front of the specified node in the PATH variable. The PATH variable must be loaded in memory.

The inserted node index number is the node\_index you specify and the inserted node value is uninitialized. The index numbers for subsequent nodes are incremented by one. You must enter the KCL> SAVE VARS command to make the change permanent.

**Examples:**

```
KCL> INSERT NODE [test_prog]weld_pth[2]
```

```
KCL> INSERT NODE weld_pth[3]
```

## C.30 LOAD ALL command

**Syntax:** LOAD ALL <file\_name> <CONVERT>

where:

file\_name : a valid file name

CONVERT : converts variables to system definition

**Purpose:** Loads a p-code and variable file from the default storage device and default directory into memory using the specified or default file name. The file types for the p-code and variable files are assumed to be .PC and .VR respectively.

If file\_name is not specified, the default program is used. If the default has not been set, then the message, Default program name not set, will be displayed.

**Examples:**

```
KCL> LOAD ALL test_prog
```

```
KCL> LOAD ALL
```

## C.31 LOAD DICT command

**Syntax:** LOAD DICT file\_name dict\_name <lang\_name>

where:

file\_name : the name of the file to be loaded

dict\_name : the name of any dictionary to be loaded. The name will be truncated to 4 characters.

lang\_name : a particular language. The available choices are ENGLISH, JAPANESE, FRENCH, GERMAN, SPANISH or DEFAULT.

**Purpose:** Loads a dictionary file from the default storage device and default directory into memory using the specified file name. The file type is assumed to be .TX.

**See Also:** [Chapter 11, DICTIONARIES AND FORMS](#)

**Examples:**

```
KCL> LOAD DICT tpaleg tpal FRENCH
```

```
KCL> LOAD DICT tpaleg tpal
```

## C.32 LOAD FORM command

---

**Syntax:** LOAD FORM <form\_name>

where:

form\_name : the name of the form to be loaded

**Purpose:** Loads the specified form, from the default storage device, into memory. A form consists of a compressed dictionary file and a variable file. If no name is specified, FORM.TX and FORM.VR are loaded.

If the specified form\_name is greater than four characters, the first two characters are not used for the dictionary name or the variable file name.

**See Also:** For more information on creating and using forms, refer to [Chapter 11, DICTIONARIES AND FORMS](#)

**Example:**

```
KCL> LOAD FORM
Loading FORM.TX with dictionary name FORM
Loading FORM.VR
```

```
KCL> LOAD FORM tpxexameg
Loading TPEXAMEG.TX with dictionary name EXAM
Loading EXAM.VR
```

## C.33 LOAD MASTER command

---

**Syntax:** LOAD MASTER <file\_name> <CONVERT>

where:

file\_name : a valid file name

CONVERT : converts variables to system definition

**Purpose:** Loads a mastering data file from the default storage device and default directory into memory using the specified or default file name. The file type is assumed to be ``.SV."

If file\_name is not specified, the default file name, SYSMAST.SV, is used.

**Example:**

```
KCL> LOAD MASTER
```

## C.34 LOAD PROGRAM command

---

**Syntax:** LOAD PROGRAM <file\_name>

where:

file\_name : a valid file name

**Purpose:** Loads a p-code file from the default storage device and default directory into memory using the specified or default file name. The file type is assumed to be .PC.

If file\_spec is not specified, the default program is used. If the default has not been set, then the message, Default program name not set, will be displayed.

**NOTE**

**For R-30iB Plus, R-30iB, and R-30iB Mate controllers, the KAREL option must be installed on the robot controller in order to load KAREL programs.**

**Examples:**

```
KCL> LOAD PROGRAM test_prog
```

```
KCL> LOAD PROG
```

## C.35 LOAD SERVO command

**Syntax:** LOAD SERVO < file\_name > <CONVERT>

where:

file\_name : a valid file name

CONVERT : converts variables to system definition

**Purpose:** Loads a servo parameter file from the default storage device and default directory into memory using the specified or default file name. The file type is assumed to be .SV.

If file\_name is not specified, the default file name, SYSSERVO.SV, is used.

**Example:**

```
KCL> LOAD SERVO
```

## C.36 LOAD SYSTEM command

**Syntax:** LOAD SYSTEM < file\_name > <CONVERT>

where:

file\_name : a valid file name

CONVERT : converts variables to system definition

**Purpose:** Loads the specified system variable file into memory, assigning values to all of the saved system variables. The default storage device and default directory are used with the specified or default file name. The file type is assumed to be .SV.

If file\_name is not specified, the default file name, SYSVARS.SV, is used.

**Examples:**

```
KCL> LOAD SYSTEM awdef
```

```
KCL> LOAD SYSTEM CONVERT
```

The following rules are applicable for system variables:

- If an array system variable that is not referenced by a program already exists when a .SV file is loaded, the size in the .SV file is used and the contents are loaded. No errors are posted.
- If an array system variable that is referenced by a program already exists when a .SV file with a LARGER size is loaded, the size in the .SV file is ignored, and NONE of the array values are loaded. The following errors are posted:

- var\_name memory not updated
- Array len creation mismatch

- If an array system variable that is referenced by a program already exists when a .SV file with a SMALLER size is loaded, the size in the .SV file is ignored but ALL the array values are loaded. No errors are posted.
- If a .SV file with a different type definition is loaded, the .SV file will stop loading and detect the error. The following errors are posted:

- Create type - var\_name failed
- Duplicate creation mismatch

- If a .SV file with a different type definition is loaded, but the CONVERT option is specified, it tries to load as much as it can. For example, the controller has a SCR\_T type which has the field \$NEW but not the field \$OLD. When an old .SV file is loaded that has \$OLD but not \$NEW, the load procedure creates the SCR\_T type based on what is in the .SV file and posts a Create type - var\_name failed, Duplicate creation mismatch error. It then creates the type SCR\_! which has the field \$OLD but not the field \$NEW. It then does a field by field copy of all of the old valid fields into the new type. Therefore, since there is not \$NEW information in the old type that field is not updated and the \$OLD information is discarded. Any fields whose types don't match are discarded from the loaded type. So if a field was changed from integer to real, the integer field in the loaded data would be discarded. Any fields that are arrays will follow the same rules as array system variables.

## C.37 LOAD TP command

**Syntax:** LOAD TP <file\_name> <OVERWRITE>

where:

file\_name : a valid file name

OVERWRITE : If specified, may overwrite a previously loaded TP program with the same name

**Purpose:** Loads a TP program from the default storage device and default directory into memory using the specified or default file name. The file type is assumed to be .TP.

If file\_name is not specified, the default program is used. If the default has not been set, then the message, Default program name not set, will be displayed.

**Examples:**

```
KCL> LOAD TP testprog
```

```
KCL> LOAD TP
```

## C.38 LOAD VARS command

**Syntax:** LOAD VARS < file\_name > <CONVERT>

where:

file\_name : a valid file name

CONVERT : converts variables to system definition

**Purpose:** Loads the specified or default variable data file from the default storage device and directory into memory. The file type is assumed to be .VR.

If file\_name is not specified, the default program is used. If the default has not been set then the message, Default program name not set, will be displayed.

**Examples:**

```
KCL> LOAD VARS test_prog
```

```
KCL> LOAD VARS
```

The following rules are applicable for array variables:

- If an array variable that is not referenced by a program already exists when a .VR file is loaded, the size in the .VR file is used and the contents are loaded. No errors are posted.
- If an array variable already exists when a program is loaded, the size in the .PC file is ignored and the program is loaded anyway. The following errors are posted:
  - var\_name PC array length ignored
  - Array len creation mismatch
- If an array variable that is referenced by a program already exists when a .VR file with a LARGER size is loaded, the size in the .VR file is ignored and NONE of the array values are loaded. The following errors are posted:
  - var\_name memory not updated
  - Array len creation mismatch
- If an array variable that is referenced by a program already exists when a .VR file with a SMALLER size is loaded, the size in the .VR file is ignored but ALL the array values are loaded. The following errors are posted:
  - var\_name array length updated
  - Array len creation mismatch

The following rules are applicable for user-defined types in KAREL programs:

- Once a type is created it can never be changed, regardless of whether a program references it or not. If all the variables referencing the type are deleted, the type will also be deleted. A new version can then be loaded.

- If a type already exists when a program with a different type definition is loaded, the .PC file will not be loaded. The following errors are posted:
  - Create type - var\_name failed
  - Duplicate creation mismatch
- If a type already exists when a .VR file with a different type definition is loaded, the .VR file will stop loading when it detects the error. The following errors are posted:
  - Create type - var\_name failed
  - Duplicate creation mismatch

## C.39 LOGOUT command

---

**Syntax:** LOGOUT

**Purpose:** Logs the current user from the KCL device out of the system. The password level reverts to the OPERATOR level. If passwords are not enabled, an error message will be displayed by KCL such as, No user currently logged in.

**Example:**

```
KCL>LOGOUT
(The alarm message: "Logout (SAM) SETUP from KCL")
KCL Username>
```

## C.40 MKDIR command

---

**Syntax:** MKDIR <device\_name>\path\_name

where:

device\_name : a valid storage device

path\_name : a subdirectory previously created on the memory card device using the mkdir command.

**Purpose:** MKDIR creates a subdirectory on the memory card (MC:) device. FANUC recommends you nest subdirectories only to 8 levels.

**Example:**

```
KCL> MKDIR mc:\test_dir
```

```
KCL> MKDIR mc:\prog_dir\tpnx_dir
```

## C.41 MOUNT command

---

**Syntax:** MOUNT device\_name

where:

device\_name : a valid storage device

**Purpose:** MOUNT indicates to the controller that a storage device is available for reading or writing data. A device must be formatted with the KCL> FORMAT command before it can be mounted successfully.

**Example:**

```
KCL> MOUNT rd:
```

## C.42 MOVE FILE command

**Syntax:** MOVE <FILE> file\_spec

where:

file\_spec : a valid file specification.

**Purpose:** Moves the specified file from one memory file device to another. The file should exist on the FROM or RAM disks. If file\_spec is a file on the FROM disk, the file is moved to the RAM disk, and vice versa.

The wildcard character (\*) can be used to replace the entire file name, the first part of the file name, the last part of the file name, or both the first and last parts of the file name. The file type can also use the wildcard in the same manner. If file\_spec specifies multiple files, then they are all moved to the other disk.

**Examples:**

```
KCL> MOVE FILE fr:*.kl
```

```
KCL> MOVE rd:.*.*
```

## C.43 PAUSE command

**Syntax:** PAUSE <( prog\_name | ALL )> <FORCE>

where:

prog\_name : the name of any KAREL or TP program which is a task

ALL : pauses all running tasks

FORCE : pauses the task even if the NOPAUSE attribute is set

**Purpose:** Pauses the specified running task. If prog\_name is not specified, the default program is used.

Execution of the current motion segment and the current program statement is completed before the task is paused.

Condition handlers remain active. If the condition handler action is NOPAUSE and the condition is satisfied, task execution resumes.

If the statement is a WAIT FOR and the wait condition is satisfied while the task is paused, the statement following the WAIT FOR is executed immediately when the task is resumed.

If the statement is a **DELAY**, timing will continue while the task is paused. If the delay time is finished while the task is paused, the statement following the **DELAY** is immediately executed when the task is resumed. If the statement is a **READ**, it will accept input even though the task is paused.

The **KCL> CONTINUE** command resumes execution of a paused task. When a task is paused, the **CYCLE START** button on the operator panel has the same effect as the **KCL> CONTINUE** command.

Examples:

```
KCL> PAUSE test_prog FORCE
```

```
KCL> PAUSE ALL
```

## C.44 PURGE command

---

**Syntax:** PURGE device\_name

where:

device\_name : the name of the memory file device to be purged

**Purpose:** Purges the specified memory file device by freeing any used blocks that are no longer needed. The device should be set to **FR:** for FROM disk, **RD:** for RAM disk, or **MF:** for both disks.

The purge operation is only necessary when the device does not have enough memory to perform an operation. The purge operation will erase file blocks that were previously used, but are no longer needed. These are called garbage blocks. The FROM disk can contain many garbage blocks if files are deleted or overwritten. The RAM disk will not normally contain garbage blocks, but they may occur when power is removed during a file copy.

The device must be mounted and no files can be open on the device or an error will be displayed.

Examples:

```
KCL> PURGE fr:
```

```
KCL> PURGE mf:
```

## C.45 PRINT command

---

**Syntax:** PRINT file\_spec

where:

file\_spec : a valid file specification

**Purpose:** Allows you to print the contents of an ASCII file to the default device.

Example:

```
KCL> PRINT testprog.kl
```

## C.46 RECORD command

**Syntax:** RECORD < [prog\_name ] > var\_name

where:

prog\_name : the name of any KAREL or TP program

var\_name : the name of any POSITION, XYZWPR, or JOINTPOS variable

**Purpose:** Records the position of the TCP and/or auxiliary or extended axes. The robot must be calibrated before the RECORD command is issued. The variable can be a system variable or a program variable that exists in memory. The position is recorded relative to the user frame of reference.

You must enter the KCL> SAVE command to permanently assign the recorded position. The Record function key, F3, under the teach pendant TEACH menu also allows you to record positions.

**Example:**

```
KCL> RECORD [paint_prog]start_pos
```

```
KCL> RECORD $GROUP[1].$uframe
```

## C.47 RENAME FILE command

**Syntax:** RENAME FILE old\_file\_spec TO new\_file\_spec

where:

old\_file\_spec : a valid file specification

new\_file\_spec : a valid file specification

**Purpose:** Changes the old\_file\_spec to the new\_file\_spec. The file will no longer exist under the old\_file\_spec. The old\_file\_spec and the new\_file\_spec must include both the file name and the file type. The same file type must be used in both file\_specs but they cannot be the same file.

Use the KCL> COPY FILE command to change the device name of a file.

**Examples:**

```
KCL> RENAME FILE test.kl TO productn.kl
```

```
KCL> RENAME FILE mycmd.cf TO yourcmd.cf
```

## C.48 RENAME VARIABLE command

**Syntax:** RENAME VARIABLE <[ prog\_name ]> old\_var\_name new\_var\_name

where:

prog\_name : the name of any KAREL or TP program

old\_var\_name : the name of any program variable

`new_var_name` : a valid program variable name

**Purpose:** Changes the `old_var_name` to the `new_var_name` in the program specified with the `old_var_name`. The variable will no longer exist under the `old_var_name`. The variable must exist in memory under the `old_var_name` in the specified program.

The `new_var_name` cannot already exist in memory. The variable still belongs to the same program. You cannot specify a `prog_name` with the `new_var_name`.

You must enter the `KCL> SAVE VARS` command to make the change permanent.

**Examples:**

```
KCL> RENAME VARIABLE [test_prog]count part_count
```

```
KCL> RENAME VAR count part_count
```

## C.49 RENAME VARS command

**Syntax:** `RENAME VARS old_prog_name new_prog_name`

where:

`old_prog_name` : the name of any KAREL or TP program

`new_prog_name` : the name of any KAREL or TP program

**Purpose:** Changes the name of the variable data associated with the `old_prog_name` to the `new_prog_name`. The variable data will no longer exist under the `old_prog_name`.

Before you use the `RENAME VARS` command, the variable data must exist in memory under the `old_prog_name`. Variable data cannot already exist in memory under the `new_prog_name`.

The command does not rename the program. To rename a KAREL program, use the `KCL> RENAME FILE` to rename the .KL file, edit the program name in the .KL file, translate the program, and load the new C file. To rename a TP program, use the **SELECT** menu.

You must enter the `KCL> SAVE VARS` command to make the change permanent.

**Example:**

```
KCL> RENAME VARS test_1 test_2
```

Use this sequence of KCL commands to copy the variable data of one program (`prog_1`) into a variable file that is then used by another program (`prog_2`):

```
LOAD VARS prog_1
RENAME VARS prog_1 prog_2
SAVE VARS prog_2
LOAD ALL prog_2
```

The effect of this sequence of commands cannot be accomplished with the `KCL> COPY FILE` command.

The name of the program to which the variable data belongs is stored in the variable file. The KCL> COPY FILE command does not change that stored program name, so the data cannot be used with another program.

## C.50 RESET command

**Syntax:** RESET

**Purpose:** Enables servo power after an error condition has shut off servo power, provided the cause of the error has been cleared. The command also clears the message line on the CRT/KB display. The error message remains displayed if the error condition still exists.

The RESET command has no effect on a program that is being executed. It has the same effect as the FAULT RESET button on the operator panel and the RESET function key on the teach pendant RESET screen.

**Example:**

```
KCL> RESET
```

## C.51 RMDIR command

**Syntax:** RMDIR <device\_name>\path\_name

where:

device\_name : a valid storage device

path\_name : a subdirectory previously created on the memory card device using the mkdir command.

**Purpose:** RMDIR deletes a subdirectory on the memory card (MC:) device. The directory must be empty before it can be deleted.

**Example:**

```
KCL> RMDIR mc:\test_dir
```

```
KCL> RMDIR mc:\test_dir\prog_dir
```

## C.52 RUN command

**Syntax:** RUN <prog\_name>

where:

prog\_name : the name of any KAREL or TP program

**Purpose:** Executes the specified program. The program must be loaded in memory. If no program is specified the default program is run. If uninitialized variables are encountered, program execution is paused.

Execution begins at the first executable line. RUN is a motion command; therefore, the device from which it is issued must have motion control. If a RUN command is issued in a command file, it is executed as a

NOWAIT command. Therefore, the statement following the RUN command will be executed immediately after the RUN command is issued without waiting for the program, specified by the RUN command, to end.

**Example:**

```
KCL> RUN test_prog
```

## C.53 RUNCF command

---

**Syntax:** RUNCF input\_file\_spec < output\_file\_spec >

where:

input\_file\_spec : a valid file specification

output\_file\_spec : a valid file specification

**Purpose:** Executes the KCL command procedure that is stored in the specified input file and displays the output to the specified output file. The input file type is assumed to be .CF. The output file type is assumed to be .LS if no file type is supplied.

If output\_file\_spec is not specified, the output will be displayed to the KCL output window.

The RUNCF command can be nested within command files up to four levels. Use %INCLUDE input\_file\_spec to include another .CF file into the command procedure. RUNCF command itself is not allowed inside a command procedure.

If the command file contains motion commands, the device from which the RUNCF command is issued must have motion control.

**See Also:** Refer to [Section 15.4, COMMAND PROCEDURES](#) for more information

**Examples:**

```
KCL> RUNCF startup output
```

```
KCL> RUNCF startup
```

## C.54 SAVE MASTER command

---

**Syntax:** SAVE MASTER < file\_name >

where:

file\_name : a valid file name

**Purpose:** Saves the mastering data file from the default storage device and default directory into memory using the specified or default file name. The file type will be .SV.

If file\_name is not specified, the default file name, SYSMAST.SV, is used.

**Example:**

```
KCL> SAVE MASTER
```

## C.55 SAVE SERVO command

**Syntax:** SAVE SERVO <file\_name>

where:

file\_name : a valid file name

**Purpose:** Saves the servo parameters into the default storage device using the specified or default file name. The file type will be .SV.

If file\_name is not specified, the default file name, SYSSERVO.SV, is used.

**Example:**

```
KCL> SAVE SERVO
```

## C.56 SAVE SYSTEM command

**Syntax:** SAVE SYSTEM <file\_name>

where:

file\_name : a valid file name

**Purpose:** Saves the system variable values into the default storage device and default directory using the specified system variable file (.SV). If you do not specify a file\_spec the default name, SYSVARS.SV, is used. For example:

```
SAVE SYSTEM file_1
```

In this case, the system variable data is saved in a variable file called file\_1.SV.

```
SAVE SYSTEM
```

In this case, the system variable data is saved in a system variable file SYSVARS.SV.

**Examples:**

```
KCL> SAVE SYSTEM file_1
```

```
KCL> SAVE SYSTEM
```

## C.57 SAVE TP command

**Syntax:** SAVE TP <file\_name> <= prog\_name>

where:

file\_name : a valid file name

prog\_name : the name of any TP program

**Purpose:** Saves the specified TP program to the specified file (.TP). If you do not specify a file\_name or a prog\_name, the default program name is used. If only a file\_name is specified, that name will also be used for prog\_name. For example:

```
SAVE TP file_1
```

In this case, the TP program file\_1 is saved in a file called file\_1.TP.

```
SAVE TP = prog_1
```

In this case, the TP program prog\_1 is saved in a file whose name is the default program name.

If you specify a program name, it must be preceded by an equal sign (=).

#### Examples:

```
KCL> SAVE TP file_1 = prog_1
```

```
KCL> SAVE TP file_1
```

```
KCL> SAVE TP = prog_1
```

```
KCL> SAVE TP
```

## C.58 SAVE VARS command

**Syntax:** SAVE VARS <file\_name> <= prog\_name>

where:

file\_name : a valid file name

prog\_name : the name of any KAREL or TP program

**Purpose:** Saves variable data from the specified program, including the currently assigned values, to the specified variable file (.VR). If you do not specify a file\_name or a prog\_name, the default program name is used. If only a file\_name is specified, that name will also be used for prog\_name. For example:

```
SAVE VARS file_1
```

In this case, the variable data for the program file\_1 is saved in a variable file called file\_1.VR.

```
SAVE VARS = prog_1
```

In this case, the variable data for prog\_1 is saved in a variable file whose name is the default program name.

If you specify a program name, it must be preceded by an equal sign (=).

Any variable data that is not saved is lost when an initial start of the controller is performed.

**Examples:**

```
KCL> SAVE VARS file_1 = prog_1
```

```
KCL> SAVE VARS file_1
```

```
KCL> SAVE VARS = prog_1
```

```
KCL> SAVE VARS
```

## C.59 SET BREAK CONDITION command

**Syntax:** SET BREAK CONDITION <prog\_name> condition\_no

where:

prog\_name : the name of any running or paused KAREL program

condition\_no : a particular condition

**Purpose:** Allows you to set a break point on the specified condition in the specified program or default program. The specified condition must already exist so the program must be running or paused. When the break point is triggered, a message will be posted to the error log and the break point will be cleared.

**Examples:**

```
KCL> SET BREAK CONDITION test_prog 1
```

```
KCL> SET BREAK COND 2
```

## C.60 SET BREAK PROGRAM command

**Syntax:** SET BREAK PROGRAM <prog\_name> brk\_pnt\_no line\_no <(PAUSE|DISPLAY|TRACE ON|TRACE OFF)>

where:

prog\_name : the name of any KAREL or TP program in memory

brk\_pnt\_no : a particular program break point

line\_no : a line number

PAUSE : task is paused when break point is executed

DISPLAY : message is displayed on the teach pendant USER menu when break point is executed

TRACE ON : trace is enabled when break point is executed

TRACE OFF : trace is disabled when break point is executed

**Purpose:** Allows you to set a break point at a specified line in the specified or default program. The specified line must be an executable line of source code. Break points will be executed before the specified line in the program. By default the task will pause when the break point is executed. DISPLAY, TRACE ON, and TRACE OFF will not pause task execution.

Break points are local only to the program in which the break points were set. For example, break point #1 can exist among one or more loaded programs with each at a unique line number. If you specify an existing break point number, the existing break point is cleared and a new one is set in the specified program at the specified line.

Break points in a program are cleared if the program is cleared from memory. You also use the KCL> CLEAR BREAK PROGRAM command to clear break points from memory.

Use the KCL> CONTINUE command or the operator panel **CYCLE START** button to resume execution of a paused program.

**Examples:**

```
KCL> SET BREAK PROGRAM test_prog 1 22 DISPLAY
```

```
KCL> SET BREAK PROG 3 30
```

---

## C.61 SET CLOCK command

---

**Syntax:** SET CLOCK 'dd-mmm-yy hh:mm'

where:

The date is specified using two numeric characters for the day, a three letter abbreviation for the month, and two numeric characters for the year; for example, 01-JAN- 00.

The time is specified using two numeric characters for the hour and two numeric characters for the minutes; for example, 12:45.

**Purpose:** Sets the date and time of the internal controller clock.

The date and time are included in directory and translator listings.

**See Also:** [Section C.74, SHOW CLOCK command](#)

**Example:**

```
KCL> SET CLOCK '02-JAN-xx 21:45'
```

---

## C.62 SET DEFAULT command

---

**Syntax:** SET DEFAULT prog\_name

where:

prog\_name : the name of any KAREL or TP program

**Purpose:** Sets the default program name to be used as an argument default for program and file names. The default program name can also be set at the teach pendant.

**See Also:** [Section 15.1.1, Default Program](#)

**Examples:**

```
KCL> SET DEFAULT test_prog
```

```
KCL> SET DEF test_prog
```

## C.63 SET GROUP command

---

**Syntax:** SET GROUP group\_no

where:

group\_no : a valid group number

**Purpose:** Sets the default group number to use in other commands.

**Example:**

```
KCL> SET GROUP 1
```

## C.64 SET LANGUAGE command

---

**Syntax:** SET LANGUAGE lang\_name

where:

lang\_name : a particular language. The available choices are ENGLISH, JAPANESE, FRENCH, GERMAN, SPANISH or DEFAULT.

**Purpose:** Sets the \$LANGUAGE system variable which determines the language to use.

**Example:**

```
KCL> SET LANG ENGLISH
```

## C.65 SET LOCAL VARIABLE command

---

**Syntax:** SET LOCAL VARIABLE var\_name <IN rout\_name> <FROM prog\_name> <task\_name> = value <{, value }>

where:

var\_name : a local variable or parameter name

rout\_name : the name of any KAREL routine

prog\_name : the name of any KAREL program

task\_name : the name of any KAREL task

value : new value for variable

**Purpose:** Assigns the specified value to the specified local variable or routine parameter. You can assign constant values or variable values, but the value must be of the data type that has been declared for the variable.

Please use the `HELP SET VAR` command for more information on assigning data types.

If the `IN` clause is omitted, the routine at the top of the stack is assumed. If the `FROM` clause is omitted, the default program is assumed. If the `task_name` is omitted, the stack of the KCL default task is searched.

#### NOTE

The file `RD: prog_name.rs` is required to obtain local variable information.

**Example:** See [Section C.83, SHOW LOCAL VARIABLE command](#)

**See Also:** [Section C.83, SHOW LOCAL VARIABLE command](#) and [Section C.100, TRANSLATE command](#)

## C.66 SET PORT command

**Syntax:** `SET PORT port_name [index] = value`

where:

`port_name[index]` : a valid I/O port

`value` : a new value for the port

**Purpose:** Assigns the specified value to a specified input or output port. `SET PORT` can be used with either physical or simulated output ports, but only with simulated input ports.

The valid ports are:

`DIN, DOUT, RDO, OPOUT, TPOUT, WDI, WDO (BOOLEAN)-AIN, AOUT, GIN, GOUT (INTEGER)`

**See Also:** [Section C.96, SIMULATE command](#) , [Section C.102, UNSIMULATE command](#) , [Chapter 16, INPUT/OUTPUT SYSTEM](#) , your application-specific *Setup and Operations Manual*, or the *OPERATOR'S MANUAL (Basic Function)* (B-83284EN).

**Example:**

```
KCL> SET PORT DOUT [1] = ON
```

```
KCL> SET PORT GOUT [2] = 255
```

```
KCL> SET PORT AIN [1] = 1000
```

## C.67 SET TASK command

**Syntax:** `SET TASK <[ prog_name ]> attr_name = value`

where:

`prog_name` : the name of any KAREL or TP program which is a task

`attr_name` : PRIORITY or TRACELEN

value : new integer value for attribute

**Purpose:** Sets the specified task attribute. PRIORITY sets the task priority. The lower the number, the higher the priority. 1 to 89 is lower than motion, but higher than the user interface. 90 to 99 is lower than the user interface. The default is 50. TRACELEN sets the trace buffer length. The default is 10 lines.

## C.68 SET TRACE command

**Syntax:** SET TRACE (OFF|ON) <[ prog\_name ]>

where:

prog\_name : the name of any KAREL or TP program loaded in memory

**Purpose:** Turns the trace function ON or OFF (default is OFF). The program statement currently being executed and its line number are stored in a buffer when TRACE is ON. TRACE should only be set to ON during debugging operations because it slows program execution. To see the trace data, SHOW TRACE command must be used.

**See Also:** [Section C.91, SHOW TRACE command](#)

## C.69 SET VARIABLE command

**Syntax:** SET VARIABLE <[ prog\_name ]> var\_name = value <{, value }>

where:

prog\_name : the name of any KAREL or TP program

var\_name : a valid program variable

value : new value for variable or a program or system variable

**Purpose:** Assigns the specified value to the specified variable. You can assign constant values or variable values, but the value must be of the data type that has been declared for the variable.

You can assign values to system variables with KCL write access, to program variables, or to standard and user-defined variables and fields. You can assign only one ARRAY element. Use brackets ([] ) after the variable name to specify an element.

Certain data types like positions and vectors might have more than one value specified.

```
KCL> SET VAR position_var = 0,0,0,0,0,0
```

The SET VARIABLE command displays the previous value of the specified variable followed by the value which you have just assigned, providing you with an opportunity to check the assignment. The DATA key on the teach pendant also allows you to assign values to variables.

When you use SET VARIABLE to define a position you can use one of the following formats:

```
KCL> SET VARIABLE var_name.X = value
KCL> SET VARIABLE var_name.Y = value
KCL> SET VARIABLE var_name.Z = value
KCL> SET VARIABLE var_name.W = value
KCL> SET VARIABLE var_name = value
```

where X,Y,Z,W,P, and R specify the location and orientation, c\_str is a string value representing configuration in terms of joint placement and turn numbers. Refer to [Section 8.1, POSITIONAL DATA](#). For example, to set X=200.0, W=60.0 and the turn numbers for axes 4 and 6 to 1 and 0 you would type the following lines:

```
KCL> SET VARIABLE var_name.X = 200
KCL> SET VARIABLE var_name.W = 60
KCL> SET VARIABLE var_name.C = '1,0'
```

You must enter the KCL>SAVE VARS command to make the changes permanent.

**See Also:** [Section 2.3, DATA TYPES](#)

**Examples:**

```
KCL> SET VARIABLE [prog1] scale = $MCR.$GENOVERRIDE
```

```
KCL> SET VAR weld_pgm.angle = 45.0
```

```
KCL> SET VAR v[2,1,3].r = -0.897
```

```
KCL> SET VAR part_array[2] = part_array[1]
```

```
KCL> SET VAR weld_pos.x = 50.0
```

```
KCL> SET VAR pth_b[3].nodepos = pth_a[3].nodepos
```

## C.70 SET VERIFY command

**Syntax:** SET VERIFY (ON | OFF)

**Purpose:** This turns the display of KCL commands ON or OFF during execution of a KCL command procedure (default is ON, meaning each command is displayed as it is executed). Only the RUNCF command is displayed when VERIFY is OFF.

## C.71 SHOW BREAK command

**Syntax:** SHOW BREAK < prog\_name >

where:

prog\_name : the name of any KAREL or TP program in memory

**Purpose:** Displays a list of program break points for the specified or default program. The following information is displayed for each break point:

- Break point number
- Line number of the break point in the program

**Examples:**

```
KCL> SHOW BREAK test_prog
```

```
KCL> SH BREAK
```

## C.72 SHOW BUILTINS command

**Syntax:** SHOW BUILTINS

**Purpose:** Displays all the softpart built-ins that are loaded on the controller.

**Example:**

```
KCL> SHOW BUILTINS
```

## C.73 SHOW CONDITION command

**Syntax:** SHOW CONDITION <prog\_name><condition\_no>

where:

prog\_name : the name of any running or paused KAREL program

condition\_no : a particular condition

**Purpose:** Displays the specified condition handler or a list of condition handlers for the specified or default program. Also displays enabled/disabled status and whether a break point is set. Condition handlers only exist when a program is running or paused.

**Examples:**

```
KCL> SHOW CONDITION test_prog
```

```
KCL> SH COND
```

## C.74 SHOW CLOCK command

**Syntax:** SHOW CLOCK

**Purpose:** Displays the current date and time of the controller clock.

**See Also:** [Section C.61, SET CLOCK command](#)

**Example:**

```
KCL> SHOW CLOCK
```

## C.75 SHOW CURPOS command

---

**Syntax:** SHOW CURPOS

**Purpose:** Displays the position of the TCP relative to the current user frame of reference with an x, y, and z location in millimeters; w, p, and r orientation in degrees; and the current configuration string. Be sure the robot is calibrated.

**Example:**

```
KCL> SHOW CURPOS
```

## C.76 SHOW DEFAULT command

---

**Syntax:** SHOW DEFAULT

**Purpose:** Shows the current default program name.

**Example:**

```
KCL> SHOW DEFAULT
```

## C.77 SHOW DEVICE command

---

**Syntax:** SHOW DEVICE device\_name:

where:

device\_name : device to be shown

**Purpose:** Shows the status of the device.

**Example:**

```
KCL> SHOW DEVICE rd:
```

## C.78 SHOW DICTS command

---

**Syntax:** SHOW DICTS

**Purpose:** Shows the dictionaries loaded in the system for the language specified in the system variable \$LANGUAGE.

**Example:**

```
KCL> SHOW DICTS
```

## C.79 SHOW GROUP command

---

**Syntax:** SHOW GROUP

**Purpose:** Shows the default group number.

**Example:**

```
KCL> SHOW GROUP
```

## C.80 SHOW HISTORY command

---

**Syntax:** SHOW HISTORY

**Purpose:** Shows the nesting information of the routine calls. To display the source lines of KAREL programs, the .KL programs must exist on the RAM disk.

**Example:**

```
KCL> SHOW HIST
```

## C.81 SHOW LANG command

---

**Syntax:** SHOW LANG

**Purpose:** Shows the language specified in the system variable *\$LANGUAGE*.

**Example:**

```
KCL> SHOW LANG
```

## C.82 SHOW LANGS command

---

**Syntax:** SHOW LANGS

**Purpose:** Shows all language currently available in the system.

**Example:**

```
KCL> SHOW LANGS
```

## C.83 SHOW LOCAL VARIABLE command

---

**Syntax:** SHOW LOCAL VARIABLE var\_name <(HEXADECIMAL|BINARY)> <IN rout\_name> <FROM prog\_name> <task\_name>

where:

var\_name : a local variable or parameter name  
 rout\_name : the name of any KAREL routine  
 prog\_name : the name of any KAREL program  
 task\_name : the name of any KAREL task

**Purpose:** Displays the name, type, and value of the specified local variable or routine parameter. Use brackets ( [ ] ) after the variable name to specify a specific ARRAY element. If you do not specify a specific element the entire variable is displayed.

If the IN clause is omitted, the routine at the top of the stack is assumed. If the FROM clause is omitted, the default program is assumed. If the task\_name is omitted, the stack of the KCL default task is searched.

#### NOTE

The file RD:prog\_name.rs is required to obtain local variable information.

**Example:** Generate a .rs file from the KAREL translator:

```
KCL> TRANS testprog RS
```

Copy the .RS file to the RD device. This is done automatically when you load the program from the KCL:

```
KCL> SET DEF testprog
KCL> LOAD PROG
Copied testprog.rs to RD:testprog.rs
```

To show local variables, the program must be running, paused, or aborted in the routine specified:

```
KCL> RUN
KCL> SHOW LOCAL VARS
KCL> SHOW LOCAL VARS IN testprog VALUES
KCL> SHOW LOCAL VAR var_1 IN rout_1 FROM testprog testtask
KCL> SHOW LOCAL VAR param_1
```

To set local variables, the program must be paused:

```
KCL> pause
KCL> set local var int_var = 12345
KCL> set local var strparam = "This is a string parameter"
```

**See Also:** [Section C.100, TRANSLATE command](#)

## C.84 SHOW LOCAL VARS command

**Syntax:** SHOW LOCAL VARS <VALUES><IN rout\_name><FROM prog\_name><task\_name>

where:

VALUES : specifies values should be displayed  
 rout\_name : the name of any KAREL routine  
 prog\_name : the name of any KAREL program  
 task\_name : the name of any KAREL task

**Purpose:** Displays a list including the name, type, and if specified, the current value of each local variable and each routine parameter.

If the IN clause is omitted, the routine at the top of the stack is assumed. If the FROM clause is omitted, the default program is assumed. If the task\_name is omitted, the stack of the KCL default task is searched

**Example:** See [Section C.83, SHOW LOCAL VARIABLE command](#)

**See Also:** [Section C.100, TRANSLATE command](#) and [Section C.83, SHOW LOCAL VARIABLE command](#)

## C.85 SHOW MEMORY command

**Syntax:** SHOW MEMORY

**Purpose:** Displays current memory status. The command displays the following status information for memory and lists each memory pool separately:

- Total number of bytes in the pool
- Available number of bytes in the pool

**Example:**

```
KCL> SHOW MEMORY
```

## C.86 SHOW PROGRAM command

**Syntax:** SHOW PROGRAM <prog\_name>

where:

prog\_name : the name of any KAREL or TP program in memory

**Purpose:** Displays the status information of the specified or default program being executed.

**Example:**

```
KCL> SHOW PROGRAM test_prog
```

```
KCL> SH PROG
```

## C.87 SHOW PROGRAMS command

**Syntax:** SHOW PROGRAMS

**Purpose:** Shows a list of programs and variable data that are currently loaded in memory.

**Examples:**

```
KCL> SHOW PROGRAMS
```

```
KCL> SH PROGS
```

## C.88 SHOW SYSTEM command

---

**Syntax:** SHOW SYSTEM <data\_type> <VALUES>

where:

data\_type : any valid KAREL data type

**Purpose:** Displays a list including the name, type, and if specified, the current value of each system variable. If you specify a **data\_type**, only the system variables of that type are listed.

**See Also:** [Section C.93, SHOW VARIABLE command](#)

**Examples:**

```
KCL> SHOW SYSTEM REAL VALUES
```

```
KCL> SH SYS
```

## C.89 SHOW TASK command

---

**Syntax:** SHOW TASK <prog\_name>

where:

prog\_name : the name of any KAREL or TP program which is a task

**Purpose:** Displays the task control data for the specified task. If prog\_name is not specified, the default program is used.

**Examples:**

```
KCL> SHOW TASK test_prog
```

```
KCL> SH TASK
```

## C.90 SHOW TASKS command

---

**Syntax:** SHOW TASKS

**Purpose:** Displays the status of all known tasks running KAREL programs or TP programs.

You may see extra tasks running that are not yours. If the teach pendant is displaying a menu that was written using KAREL, such as **Program Adjustment** or **Setup Passwords**, you will see the status for this task also.

**Examples:**

```
KCL> SHOW TASKS
```

## C.91 SHOW TRACE command

**Syntax:** SHOW TRACE <prog\_name>

where:

prog\_name : the name of any KAREL or TP program which is a task

**Purpose:** Shows all the program statements and line numbers that have been executed since TRACE has been turned on.

The number of lines that are shown depends on the trace buffer length, which can be set with the SET\_TASK command or the SET\_TSK\_ATTR built-in routine. To display the source lines of KAREL programs, the .KL files must exist on the RAM disk.

**See Also:** [Section C.68, SET TRACE command](#)

**Example:**

```
KCL> SHOW TRACE
```

## C.92 SHOW TYPES command

**Syntax:** SHOW TYPES <prog\_name><FIELDS>

where:

prog\_name : the name of any KAREL or TP program

FIELDS : specifies fields should be displayed

**Purpose:** Displays a list including the name, type, and if specified, the fields of each user-defined type in the specified or default program. The actual array dimensions and string sizes are not shown.

**See Also:** [Section C.94, SHOW VARS command](#), [Section C.93, SHOW VARIABLE command](#)

**Examples:**

```
KCL> SHOW TYPES test_prog FIELDS
```

```
KCL> SH TYPES
```

## C.93 SHOW VARIABLE command

**Syntax:** SHOW VARIABLE <[ prog\_name ]> var\_name<( HEXADECIMAL | BINARY)>

where:

prog\_name : the name of any KAREL or TP program

`var_name` : a valid program variable

**Purpose:** Displays the name, type, and value of the specified variable.

You can display the values of system variables that allow KCL read access or the values of program variables. Use brackets ( [ ] ) after the variable name to specify a specific ARRAY element. If you do not specify a specific element the entire variable is displayed.

**See Also:** [Section C.94, SHOW VARS command](#) , [Section C.88, SHOW SYSTEM command](#)

**Examples:**

```
KCL> SHOW VARIABLE $UTOOL
```

```
KCL> SH VAR [test_prog]group_mask HEX
```

```
KCL> SH VAR [test_prog]group_mask BINARY
```

```
KCL> SH VAR weld_pth[3]
```

## C.94 SHOW VARS command

**Syntax:** `SHOW VARS <prog_name><VALUES>`

where:

`prog_name` : the name of any KAREL or TP program

`VALUES` : specifies values should be displayed

**Purpose:** Displays a list including the name, type and, if specified, the current value of each variable in the specified or default program.

**See Also:** [Section C.93, SHOW VARIABLE command](#) , [Section C.88, SHOW SYSTEM command](#) , [Section C.92, SHOW TYPES command](#)

**Example:**

```
KCL> SHOW VARS test_prog VALUES
```

```
KCL> SH VARS
```

## C.95 SHOW data\_type command

**Syntax:** `SHOW data_type <prog_name><VALUES>`

where:

`data_type` : any valid KAREL data type

`prog_name` : the name of any KAREL or TP program

`VALUES` : specifies values should be displayed

**Purpose:** Displays a list of variables in the specified or default program (prog\_name) of the specified data type (data\_type). The list includes the name, type, and if specified, the current value of each variable.

**See Also:** [Section C.94, SHOW VARS command](#), [Section C.93, SHOW VARIABLE command](#)

**Examples:**

```
KCL> SHOW REAL test_prog VALUES
```

```
KCL> SH INTEGER
```

## C.96 SIMULATE command

**Syntax:** SIMULATE port\_name[index]< = value >

where:

port\_name[index] : a valid I/O port

value : a new value for the port

**Purpose:** Simulating I/O allows you to test a program that uses I/O. Simulating I/O does not actually send output signals or receive input signals.

 **WARNING**

Depending on how signals are used, simulating signals might alter program execution.  
Do not simulate signals that are set up for safety checks. If you do, you could injure personnel or damage equipment.

When simulating a port value, you can specify its initial simulated value or allow the initial value to be the same as the physical port value. If no value is specified, the current physical port value is used.

The valid ports are:

DIN, DOUT, WDI, WDO (BOOLEAN) AIN, AOUT, GIN, GOUT (INTEGER)

**See Also:** [Section C.102, UNSIMULATE command](#)

**Examples:**

```
KCL> SIMULATE DIN[17]
```

```
KCL> SIM DIN[1] = ON
```

```
KCL> SIM AIN[1] = 100
```

## C.97 SKIP command

**Syntax:** SKIP <prog\_name>

where:

prog\_name : the name of any KAREL or TP program which is a task

**Purpose:** Skips execution of the current statement in the specified task. If prog\_name is not specified, the default program is used. It has no effect when a task is running or when the system is in a READY state.

Entire motion statements are skipped with this command. You cannot skip single motion segments. The KCL> CONTINUE command resumes execution of the paused task with the statement following the last skipped statement. END statements cannot be skipped.

If you skip the last RETURN statement in a function routine, there is no way to return the value of the function to the calling program. Therefore, when executing the END statement of the routine, the task will abort.

If you skip into a FOR loop, you have skipped the statement that initializes the loop counter. When the ENDFOR statement is executed the program will try to remove the loop counter from the stack. If the FOR loop was nested in another FOR loop, the loop counter for the previous FOR loop will be removed from the stack, causing potentially invalid results. If the FOR loop was not nested, a stack underflow error will occur, causing the task to abort.

READ, MOVE, DELAY, WAIT FOR, and PULSE statements can be paused after they have begun execution. In these cases, when the task is resumed, execution of the paused statement must be finished before subsequent statements are executed. Subsequent skipped statements will not be executed. In particular, READ and WAIT FOR statements often require user intervention, such as entering data, before statement execution is completed.

Step mode operation and step mode type have no effect on the KCL> SKIP command.

#### Examples:

```
KCL> SKIP test_prog
```

```
KCL> SKIP
```

## C.98 STEP OFF command

**Syntax:** STEP OFF

**Purpose:** Disables single stepping for the program in which it was enabled.

#### Example:

```
KCL> STEP OFF
```

## C.99 STEP ON command

**Syntax:** STEP ON <prog\_name>

where:

prog\_name : the name of any KAREL or TP program which is a task

**Purpose:** Enables single stepping for the specified or default program.

**Examples:**

```
KCL> STEP ON test_prog
```

```
KCL> STEP ON
```

## C.100 TRANSLATE command

---

**Syntax:** TRANSLATE <file\_spec> <DISPLAY> <LIST> <RS>

where:

file\_spec : a valid file specification

DISPLAY : display source during translation

LIST : create listing file

RS : create routine stack (.rs) file for local var access

**Purpose:** Translates KAREL source code (.KL type files) into p-code (.PC type files), which can be loaded into memory and executed.

Translation of a program can be canceled using the **CANCEL COMMAND** key, **CTRL-C**, or **CTRL-Y** on the CRT/KB.

**Examples:**

```
KCL> TRANSLATE testprog DISPLAY LIST
```

```
KCL> TRAN
```

## C.101 TYPE command

---

**Syntax:** TYPE file\_spec

where:

file\_spec : a valid file specification

**Purpose:** This command allows you to display the contents of the specified ASCII file on the CRT/KB. You can specify any type of ASCII file.

**Examples:**

```
KCL> TYPE rd:testprog.kl
```

```
KCL> TYPE testprog.kl
```

## C.102 UNSIMULATE command

---

**Syntax:** UNSIMULATE ( port\_name[index] | ALL )

where:

port\_name[index] : a valid I/O port

ALL : all simulated I/O ports

**Purpose:** Discontinues simulation of the specified input or output port. When a port is unsimulated, the physical value replaces the simulated value.

 **WARNING**

Depending on how signals are used, unsimulating signals might alter program execution or activate peripheral equipment. Do not unsimulate a signal unless you are sure of the result. If you do, you could injure personnel or damage equipment.

If you specify ALL instead of a particular port, simulation on all the simulated ports is discontinued.

The valid ports are:

DIN, DOUT, WDI, WDOAIN, AOUT, GIN, GOUT

**See Also:** [Section C.96, SIMULATE command](#)

Examples:

```
KCL> UNSIMULATE DIN[17]
```

```
KCL> UNSIM ALL
```

## C.103 WAIT command

**Syntax:** WAIT <prog\_name> ( DONE | PAUSE )

where:

prog\_name : the name of any KAREL or TP program which is a task

DONE : specifies that the command procedure wait until execution of the current task is completed or aborted

PAUSE : specifies that the command procedure wait until execution of the current task is paused, completed, or aborted.

**Purpose:** Defers execution of the commands that follow the KCL> WAIT command in a command procedure until a task pauses or completes execution.

The command procedure waits until the condition specified with the DONE or PAUSE argument is met.

**See Also:** [Section 15.4, COMMAND PROCEDURES](#)

**Example:** The following is an example of an executable command procedure:

```
> SET DEF testprog
> LOAD ALL
> RUN -- execute program
> WAIT PAUSE
> SHOW CURPOS -- display position of TCP when program pauses
> CONTINUE
```

```
> WAIT DONE
> CLEAR ALL YES -- clear after execution
```

EFFMAN  
QUIRIONR

## D CHARACTER CODES

This appendix lists the ASCII numeric decimal codes and their corresponding ASCII, Multinational, graphic, and European characters as implemented on the KAREL system. The ASCII character set is the default character set for the KAREL system. Use the CHR Built-In Function, in Appendix A, to access the Multinational and Graphics character sets.

**Table D (a) ASCII Character Codes**

Decimal Code	Character Value						
000	(NUL)	032	SP	064	@	096	'
001	(SOH)	033	!	065	A	097	a
002	(STX)	034	"	066	B	098	b
003	(ETX)	035	#	067	C	099	c
004	(EOT)	036	\$	068	D	100	d
005	(ENQ)	037	%	069	E	101	e
006	(ACK)	038	&	070	F	102	f
007	(BEL)	039	'	071	G	103	g
008	(BS)	040	(	072	H	104	h
009	(HT)	041	)	073	I	105	i
010	(LF)	042	*	074	J	106	j
011	(VT)	043	+	075	K	107	k
012	(FF)	044	'	076	L	108	l
013	(CR)	045	-	077	M	109	m
014	(SO)	046	.	078	N	110	n
015	(SI)	047	/	079	O	111	o
016	(DLE)	048	0	080	P	112	p
017	(DC1)	049	1	081	Q	113	q
018	(DC2)	050	2	082	R	114	r
019	(DC3)	051	3	083	S	115	s
020	(DC4)	052	4	084	T	116	t
021	(NAK)	053	5	085	U	117	u
022	(SYN)	054	6	086	V	118	v
023	(ETB)	055	7	087	W	119	w
024	(CAN)	056	8	088	X	120	x
025	(EM)	057	9	089	Y	121	y
026	(SUB)	058	:	090	Z	122	z

Decimal Code	Character Value						
027	(ESC)	059	;	091	[	123	{
028	(FS)	060	<	092	\	124	
029	(GS)	061	=	093	]	125	}
030	(RS)	062	>	094	^	126	~
031	(US)	063	?	095	—	127	(DEL)

**Table D (b) Special ASCII Character Codes**

Decimal Code	Character Value	Decimal Code	Character Value
128	Clear window	154	Turn Multinational mode on
129	Clear to end of line	155 48	Foreground color black
130	Clear to end of window	155 49	Foreground color red
131	Set cursor position	155 50	Foreground color green
132	Carriage return	155 51	Foreground color yellow
133	Line feed	155 52	Foreground color blue
134	Reverse line feed	155 53	Foreground color magenta
135	Carriage return & line feed	155 54	Foreground color cyan
136	Back Space	155 55	Foreground color white
137	Home cursor in window	155 127	Foreground color default
138	Blink video attribute	156 48	Background color black
139	Reverse video attribute	156 49	Background color red
140	Bold video attribute	156 50	Background color green
141	Underline video attribute	156 51	Background color yellow
142	Wide video size	156 52	Background color blue
143	Normal video attribute	156 53	Background color magenta
146	Turn Graphics mode on	156 54	Background color cyan
147	Turn ASCII mode on	156 55	Background color white
148	High/wide video size	156 127	Background color default
153	Normal video size		

**Table D (c) Multinational Character Codes**

Decimal Codes	Character Value						
000		032		064	À	096	à
001		033	í	065	Á	097	á
002		034	©	066	Â	098	â

Decimal Codes	Character Value						
003		035	£	067	Ā	099	ā
004	(IND)	036		068	Ä	100	ä
005	(NEL)	037	¥	069	Å	101	å
006	(SSA)	038		070	Æ	102	æ
007	(ESA)	039	§	071	Ç	103	ç
008	(HTS)	040	¤	072	È	104	è
009	(HTJ)	041	©	073	É	105	é
010	(VTS)	042	ₐ	074	Ê	106	ê
011	(PLD)	043	«	075	Ë	107	ë
012	(PLU)	044		076	Ì	108	ì
013	(RI)	045		077	Í	109	í
014	(SS2)	046		078	Î	110	î
015	(SS3)	047		079	Ï	111	ï
016	(DCS)	048	0	080		112	
017	(PU1)	049	±	081	Ñ	113	ñ
018	(PU2)	050	2	082	Ò	114	ò
019	(STS)	051	3	083	Ó	115	ó
020	(CCH)	052		084	Ô	116	ô
021	(MW)	053	µ	085	Õ	117	õ
022	(SPA)	054	¶	086	Ö	118	ö
023	(EPA)	055	•	087	Œ	119	œ
024		056		088	Ø	120	ø
025		057	¡	089	Ù	121	ù
026		058		090	Ú	122	ú
027	(CSI)	059	»	091	Û	123	û
028	(ST)	060	¼	092	Ü	124	ü
029	(OSC)	061	½	093	Ý	125	ÿ
030	(PM)	062		094		126	
031	(APC)	063	¿	095	ß	127	

Table D (d) Graphics Character Codes (not available in R-30iB)

Decimal Codes	Character Value						
000	(NUL)	032	SP	064	@	096	♦
001	(SOH)	033	!	065	A	097	¤
002	(STX)	034	"	066	B	098	H_T

Decimal Codes	Character Value						
003	(ETX)	035	#	067	C	099	F_F
004	(EOT)	036	\$	068	D	100	C_R
005	(ENQ)	037	%	069	E	101	L_F
006	(ACK)	038	&	070	F	102	f
007	(BEL)	039	'	071	G	103	±
008	(BS)	040	(	072	H	104	N_L
009	(HT)	041	)	073	I	105	V_T
010	(LF)	042	*	074	J	106	J
011	(VT)	043	+	075	K	107	Γ
012	(FF)	044	'	076	L	108	Γ
013	(CR)	045	-	077	M	109	L
014	(SO)	046	.	078	N	110	+
015	(SI)	047	/	079	O	111	-
016	(DLE)	048	0	080	P	112	-
017	(DC1)	049	1	081	Q	113	-
018	(DC2)	050	2	082	R	114	-
019	(DC3)	051	3	083	S	115	—
020	(DC4)	052	4	084	T	116	⊣
021	(NAK)	053	5	085	U	117	⊣
022	(SYN)	054	6	086	V	118	⊥
023	(ETB)	055	7	087	W	119	⊤
024	(CAN)	056	8	088	X	120	
025	(EM)	057	9	089	Y	121	≤
026	(SUB)	058	:	090	Z	122	≥
027	(ESC)	059	;	091	[	123	□
028	(FS)	060	<	092		124	≠
029	(GS)	061	=	093	]	125	£
030	(RS)	062	>	094	^	126	•
031	(US)	063	?	095	(blank)	127	(DEL)

Table D (e) Teach Pendant Input Codes

Code	Value	Code	Value
0	48	174	USER KEY 2
1	49	175	USER KEY 3
2	50	176	USER KEY 4

Code	Value	Code	Value
3	51	177	USER KEY 5
4	52	178	USER KEY 6
5	53	185	FWD
6	54	186	BWD
7	55	187	COORD
8	56	188	+X
9	57	189	+Y
128	PREV	190	+Z
129	F1	191	+X rotation
131	F2	192	+Y rotation
132	F3	193	+Z rotation
133	F4	194	-X
134	F5	195	-Y
135	NEXT	196	-Z
143	SELECT	197	-X rotation
144	MENUS	198	-Y rotation
145	EDIT	199	-Z rotation
146	DATA	210	USER KEY
147	FCTN	212	Up arrow
148	ITEM	213	Down arrow
149	+%	214	Right arrow
150	-%	215	Left arrow
151	HOLD	147	DEADMAN switch, left
152	STEP	248	DEADMAN switch, right
153	RESET	249	ON/OFF switch
173	USER KEY 1	250	EMERGENCY STOP

Table D (f) European Character Codes

Code	Value	Code	Value	Code	Value
192	A`	213	O~	234	e^
193	A`	214	O:	235	e:
194	A^	215	OE	236	i`
195	A~	216	O/	237	i`
196	A:	217	U`	238	i^
197	Ao	218	U`	239	i:
198	AE	219	U^	240	
199	CC	220	U:	241	n~

Code	Value	Code	Value	Code	Value
200	E`	221	Y:	242	o`
201	E`	222		243	o`
202	E^	223	Bb	244	o^
203	E:	224	a`	245	o~
204	I`	225	a`	246	o:
205	I`	226	a^	247	oe
206	I^	227	a~	248	
207	I:	228	a:	249	u`
208		229	ao	250	u`
209	N~	230	ae	251	u^
210	O`	231		252	u:
211	O`	232	e`	253	y:
212	O^	233	e`	254	

A^ = A with ^ on top

A` = A with ` on top

Ao = A with o on top

A~ = A with ~ on top

A: = A with .. on top

AE = A and E run together

OE = A and E run together

Bb = Beta

**Table D (g) Graphics Characters**

Decimal Value	ASCII Character	Graphic Character
97	a	solid box
102	f	diamond
103	g	plus/minus
106	j	lower-right box corner
107	k	upper-right box corner
108	l	upper-left box corner
109	m	lower-left box corner
110	n	intersection lines
111	o	pixel row 1 horizontal line
112	p	pixel row 2 horizontal line
113	q	pixel row 3 horizontal line
114	r	pixel row 4 horizontal line

Decimal Value	ASCII Character	Graphic Character
115	s	pixel row 5 horizontal line
116	t	T from right
117	u	T from left
119	v	T from above
119	w	T from below
120	x	Vertical Line
121	y	Less than or equal
122	z	Greater than or equal
123	{	Pi
124		Not equal
125	}	British pound symbol

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR

## E SYNTAX DIAGRAMS

KAREL syntax diagrams use the following symbols:

- Rectangle



A rectangle encloses elements that are defined in another syntax diagram or in accompanying text.

- Oval



An oval encloses KAREL reserved words that are entered exactly as shown.

- Circle



A circle encloses special characters that are entered exactly as shown.

- Dot



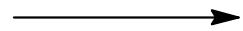
A dot indicates a mandatory line-end (; or ENTER key) before the next syntax element.

- Caret



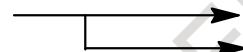
A caret indicates an optional line-end.

- Arrows



Arrows indicate allowed paths and the correct sequence in a diagram.

- Branch



Branches indicate optional paths or sequences.

**Figure E (a) Syntax Diagram Symbols**

## PROGRAM—module definition

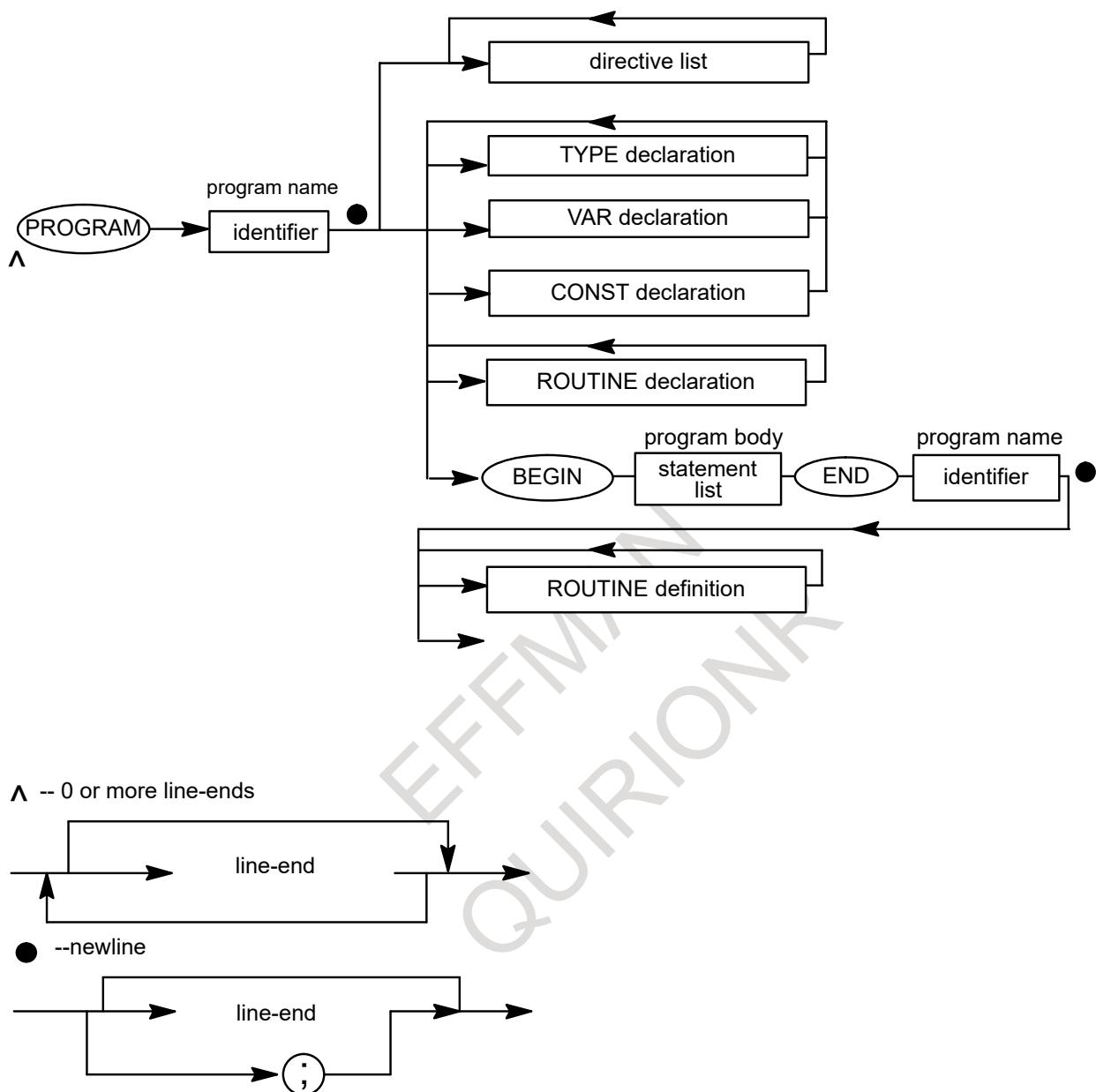
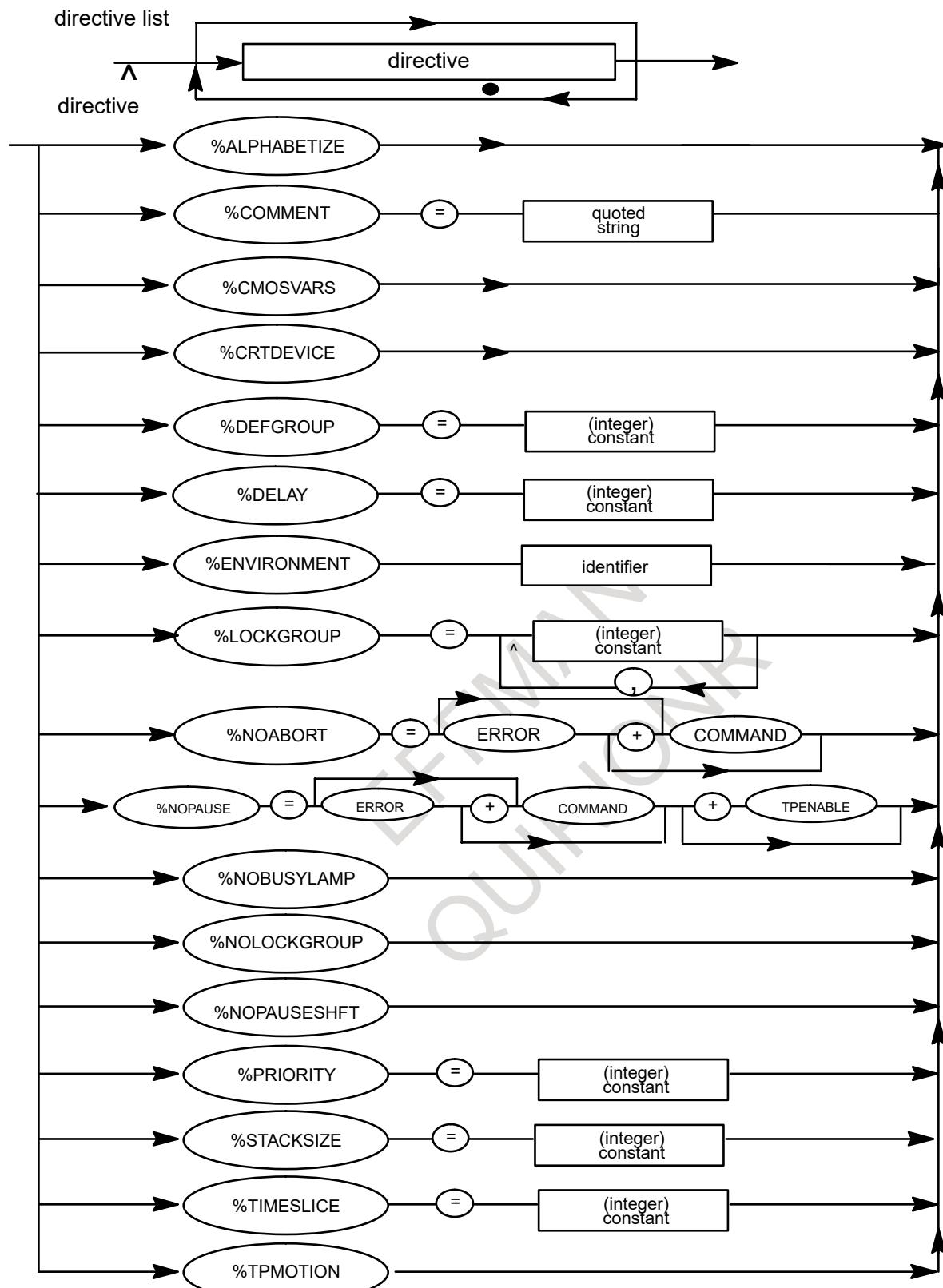
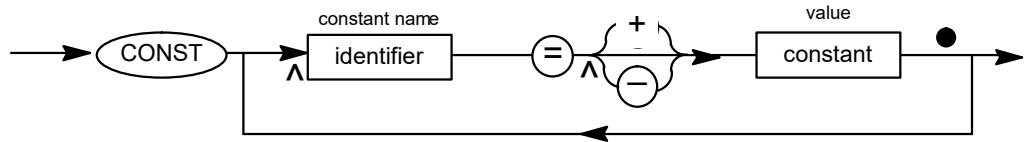


Figure E (b) PROGRAM – Module Definition

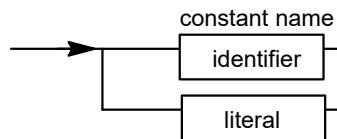


### **Figure E (c) Directive List**

CONST --constant declaration



constant



TYPE -- type declaration

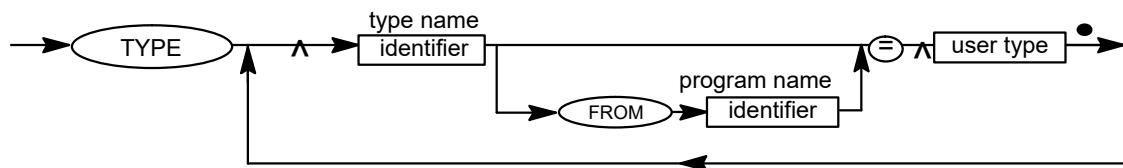
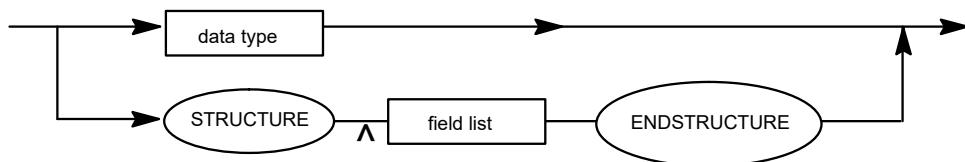
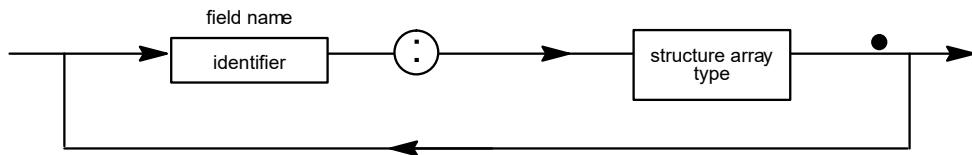


Figure E (d) Constant and Type Declarations

user type



field list



structure array type

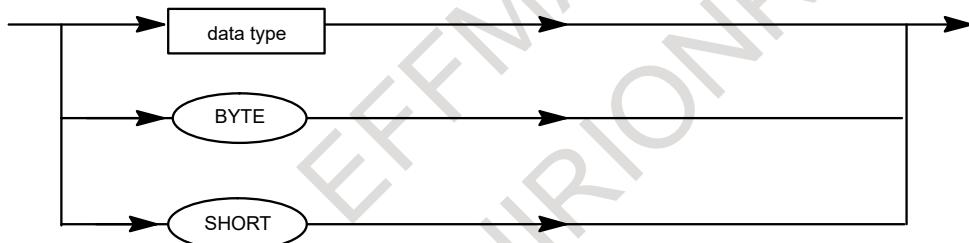


Figure E (e) User Type, Field List, Structure Array Type

data type

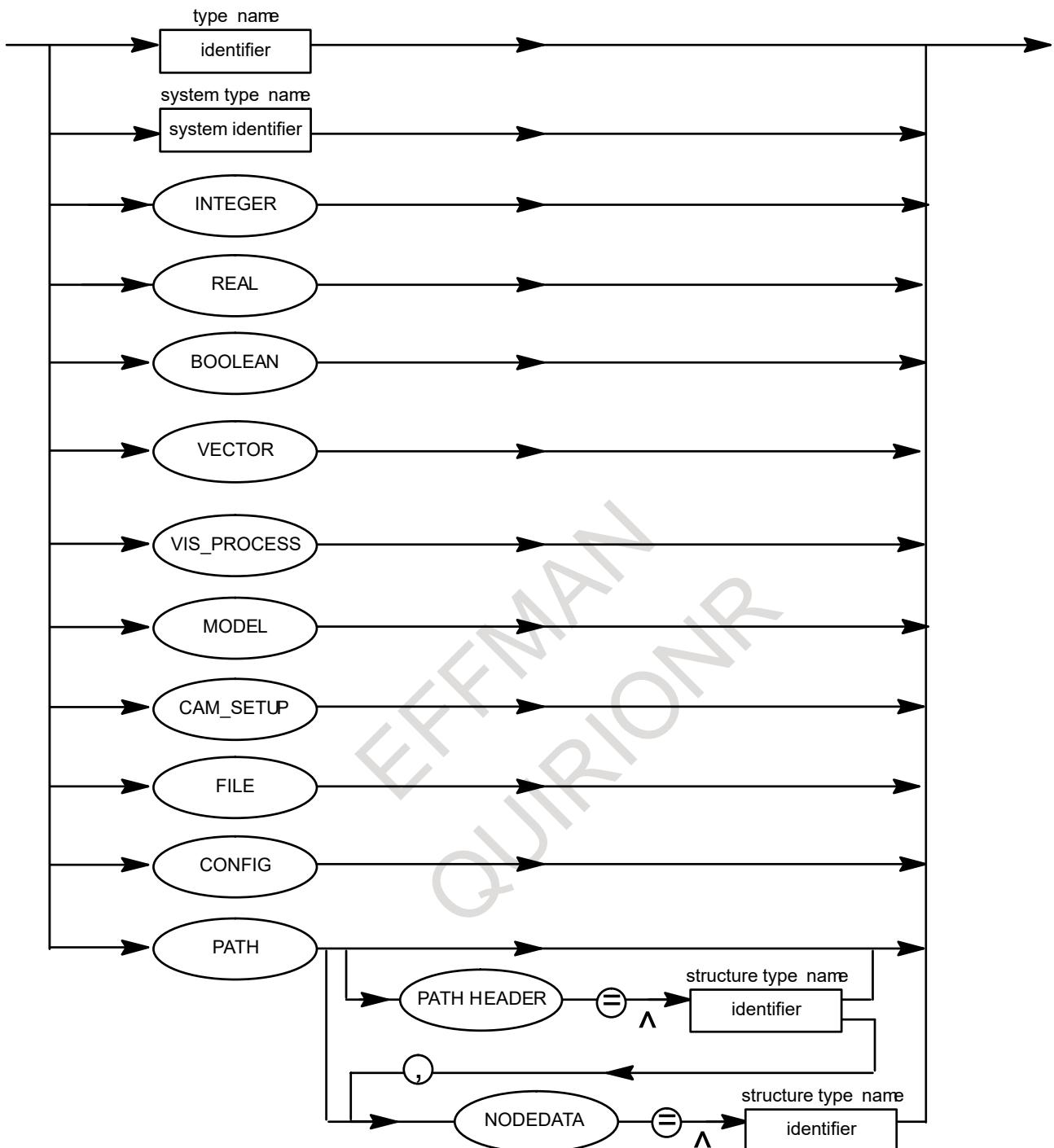


Figure E (f) Data Type

data type continued

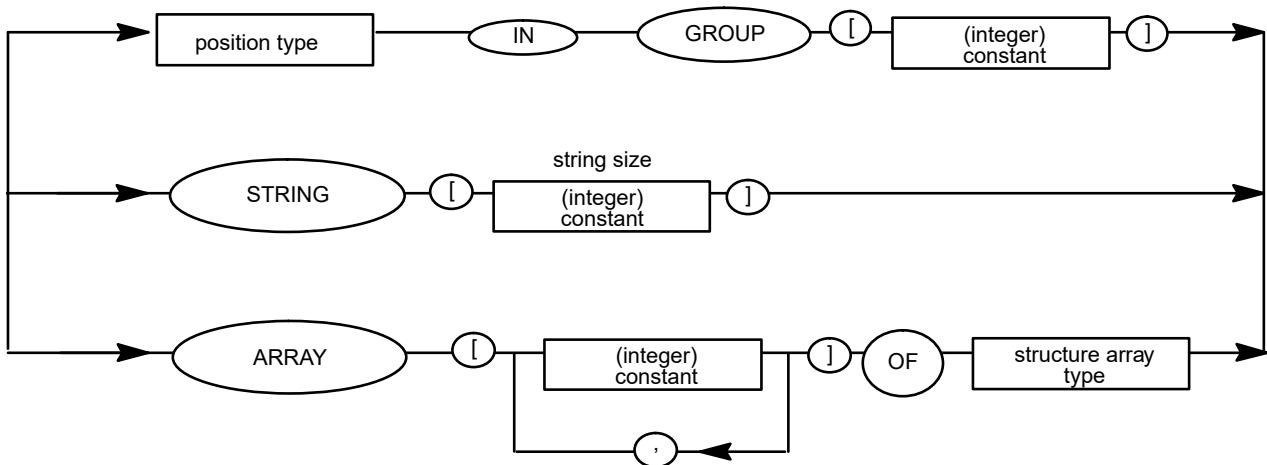


Figure E (g) Data Type, continued

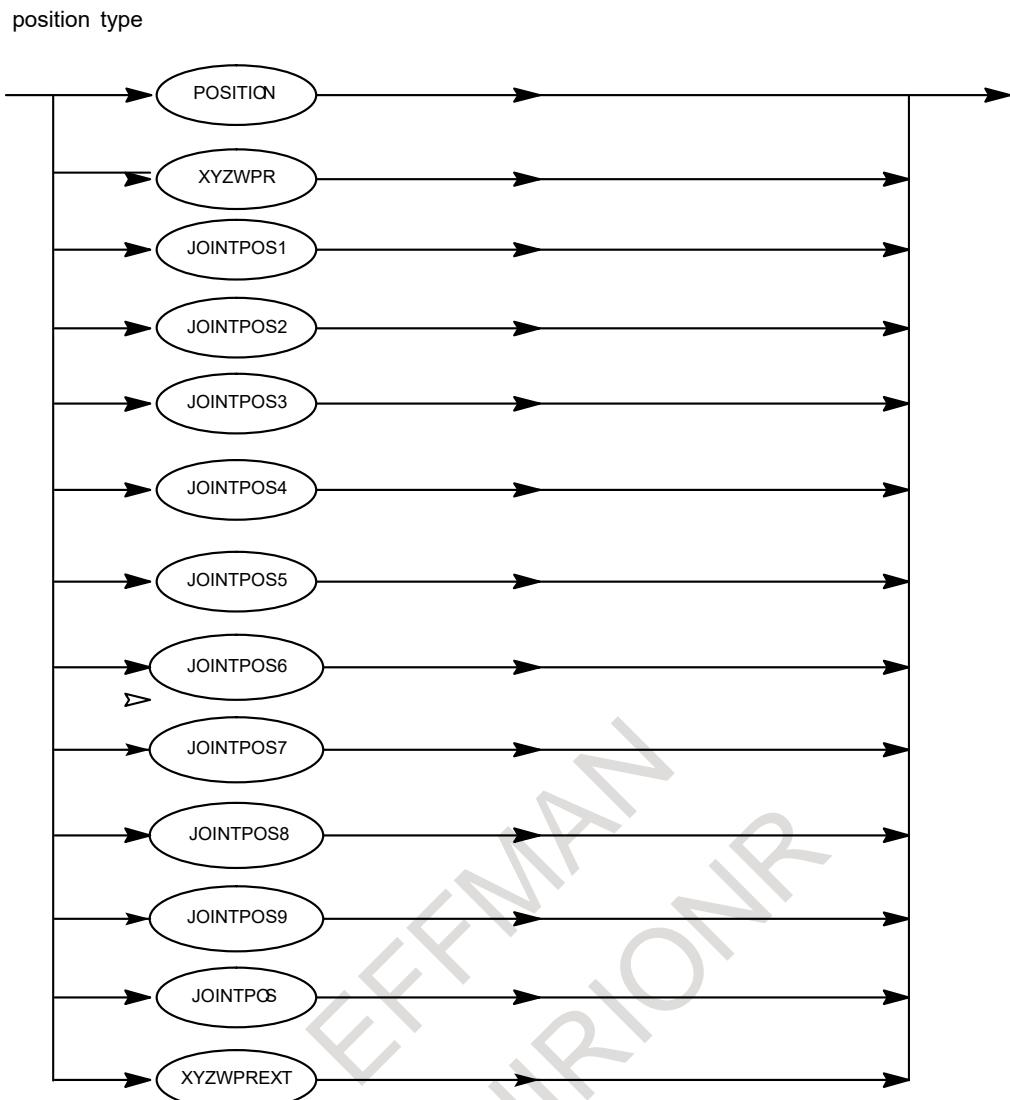
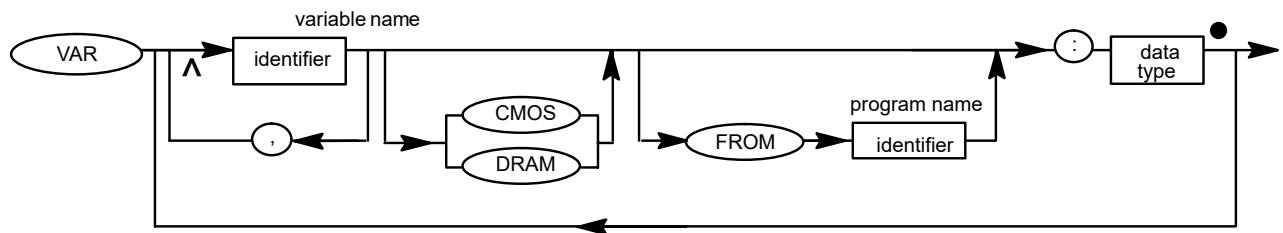


Figure E (h) Position Type

VAR -- variable declaration



ROUTINE -- routine declaration

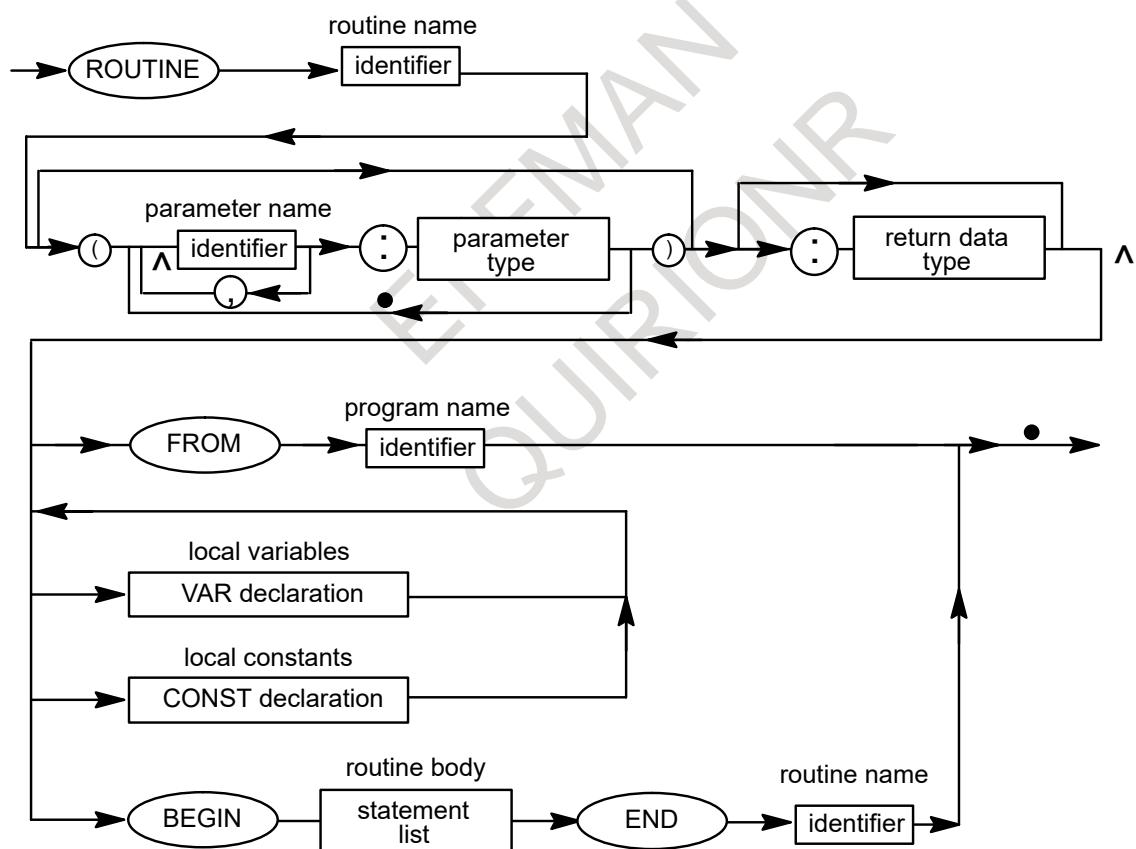


Figure E (i) Variable Declaration, Routine Declaration

return data type

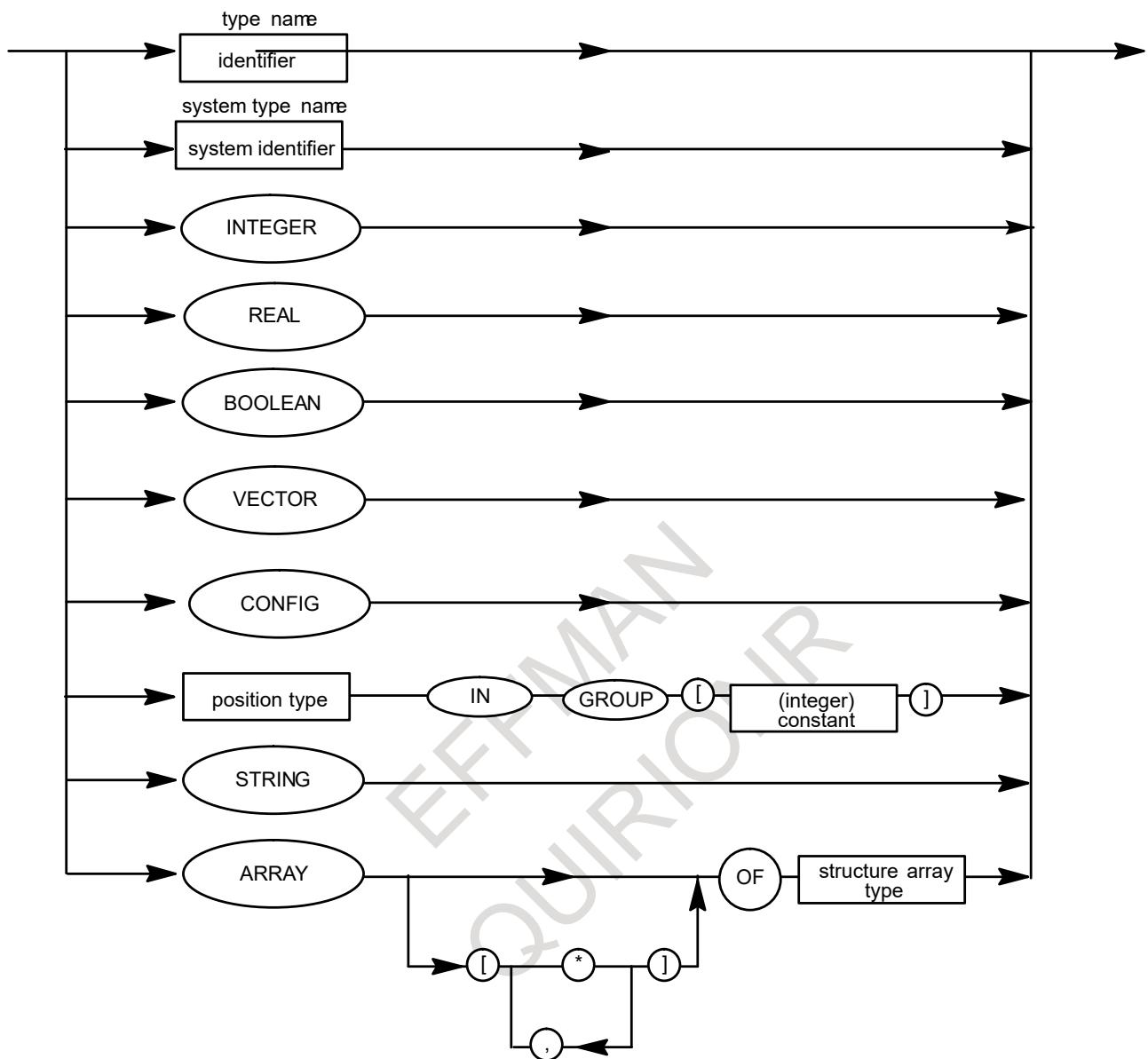
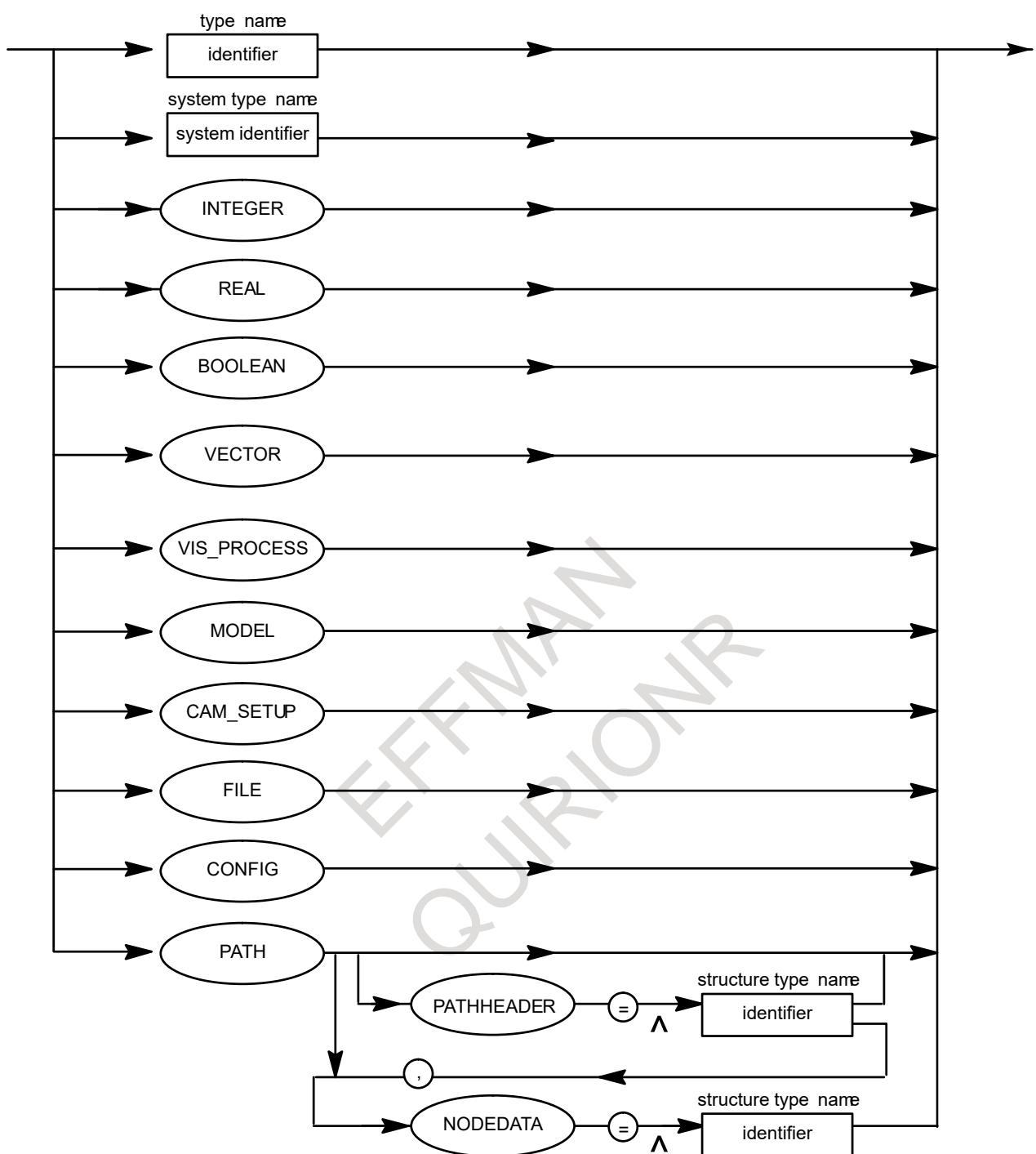


Figure E (j) Return Data Type

parameter type



**Figure E (k) Parameter Type**

parameter type continued

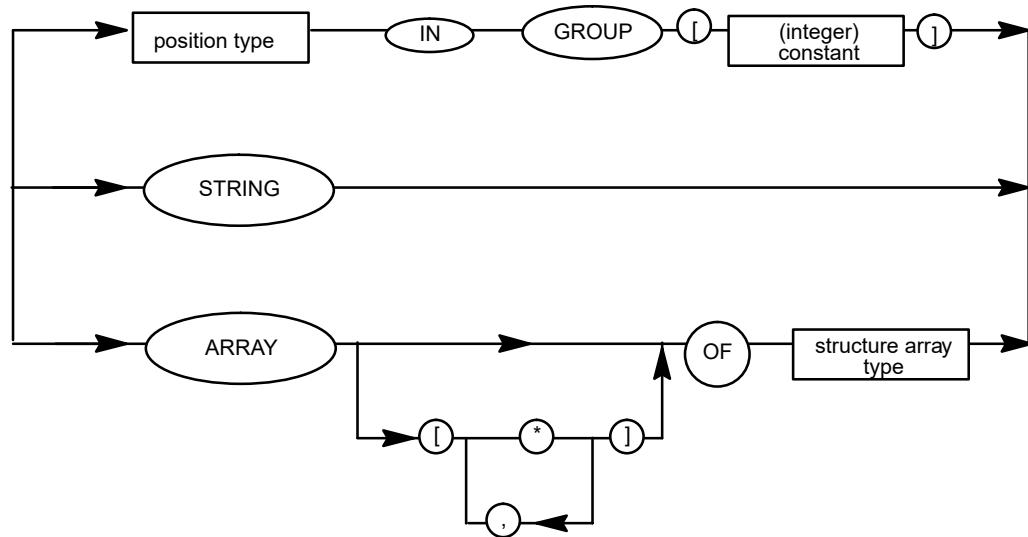


Figure E (I) Parameter Type, continued

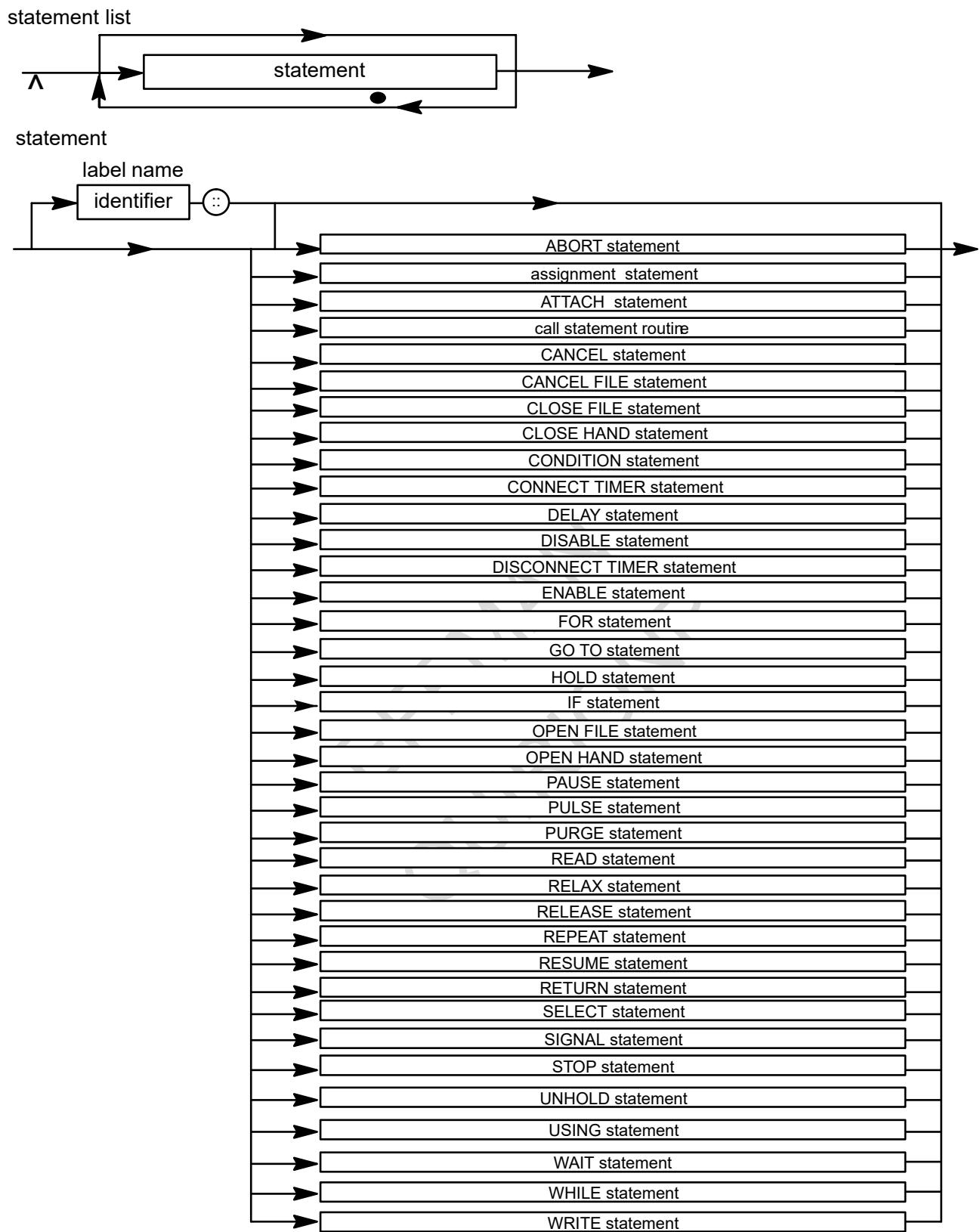
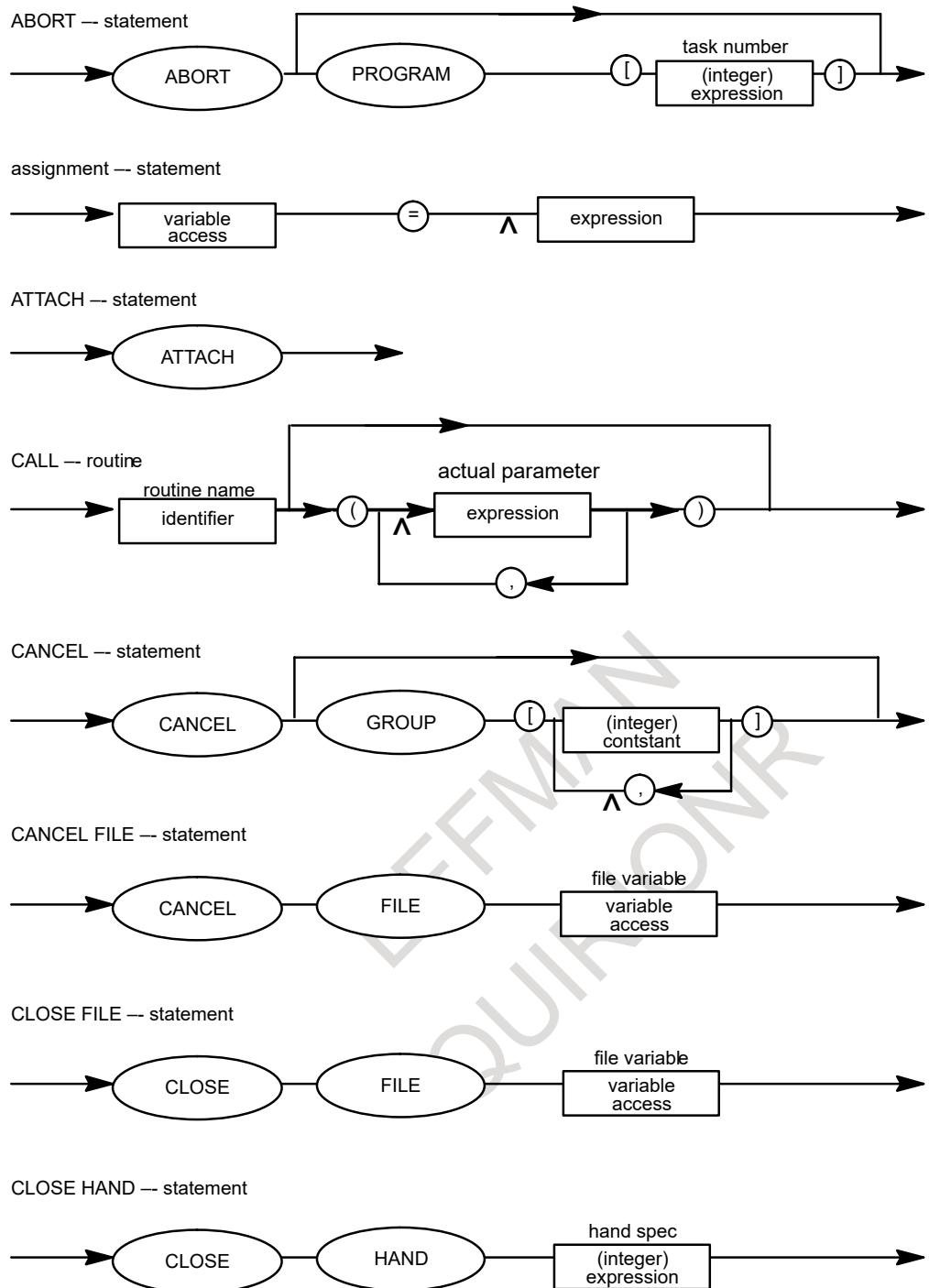
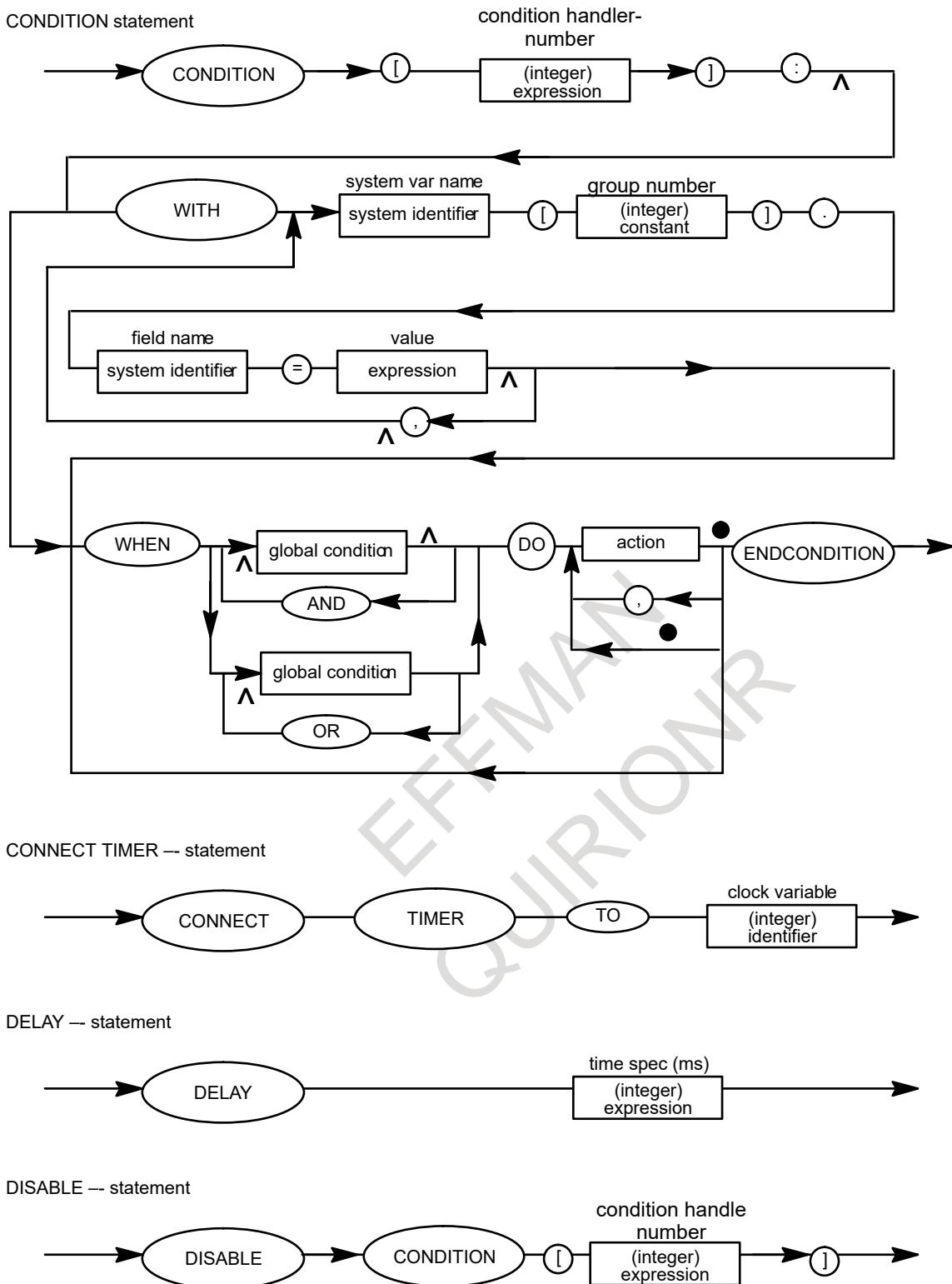


Figure E (m) Statement List



**Figure E (n) ABORT, ATTACH, CALL, CANCEL, CANCEL FILE, CLOSE FILE, CLOSE HAND Statements**

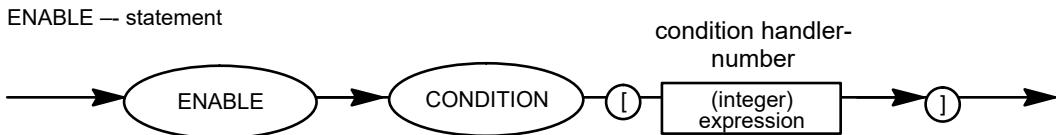


**Figure E (o) CONDITION, CONNECT TIMER, DELAY, DISABLE Statements**

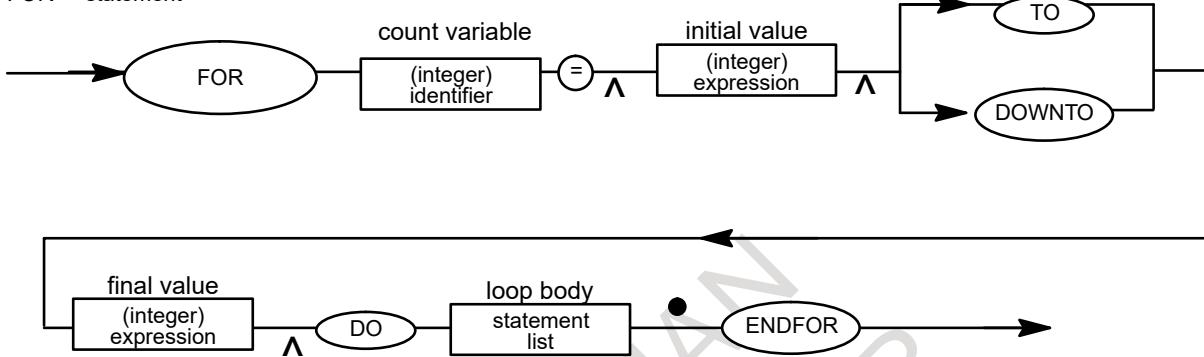
DISCONNECT TIMER -- statement



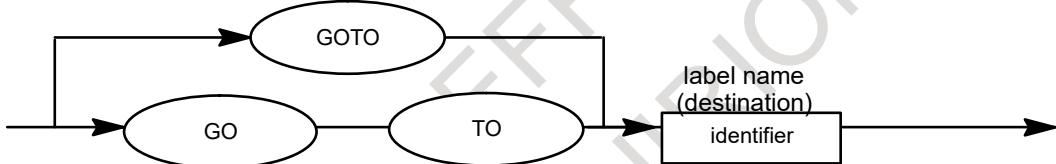
ENABLE -- statement



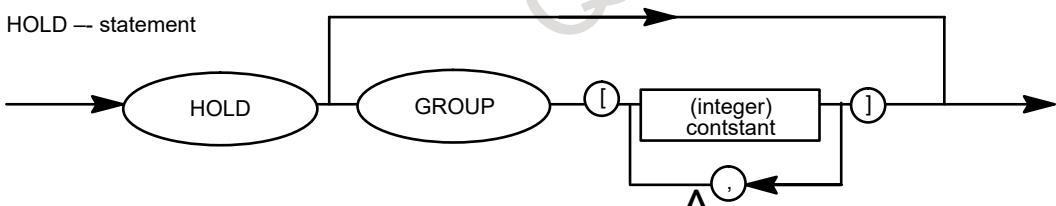
FOR -- statement



GO TO -- statement



statement label



IF THEN -- statement

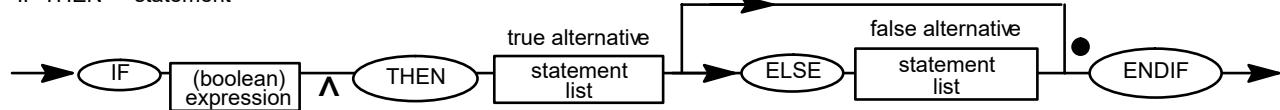
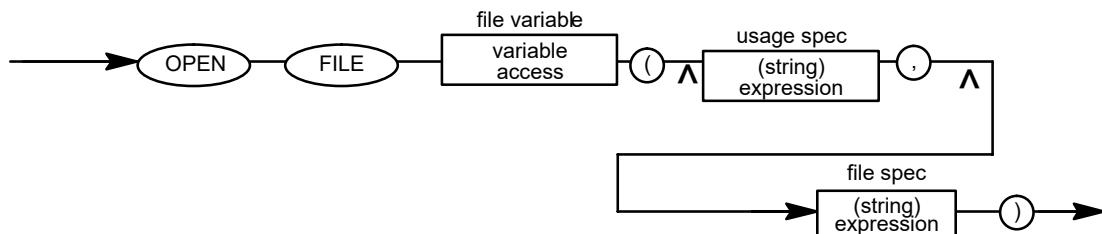
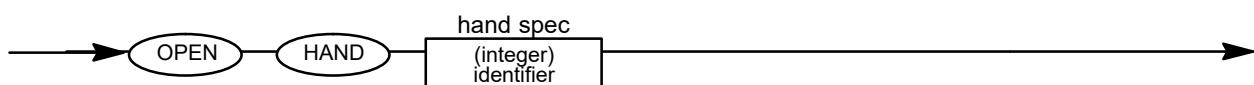


Figure E (p) DISCONNECT TIMER, ENABLE, FOR, GO TO, HOLD, IF THEN Statements

OPEN FILE -- statement



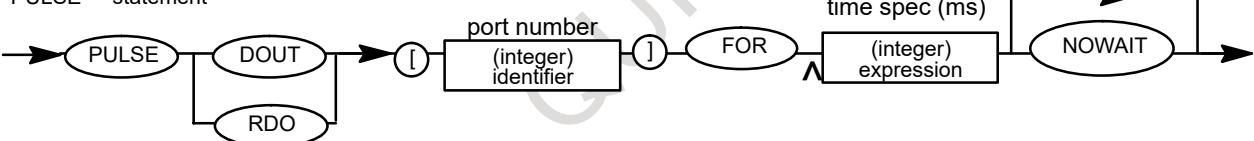
OPEN HAND -- statement



PAUSE -- statement



PULSE -- statement



PURGE -- statement

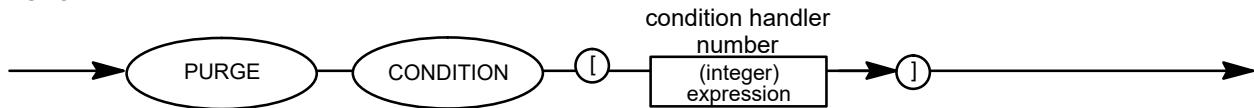
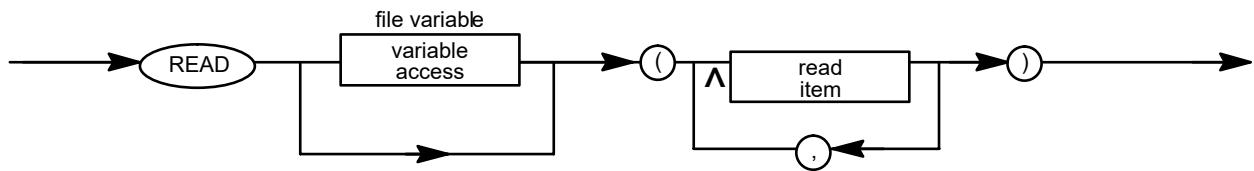
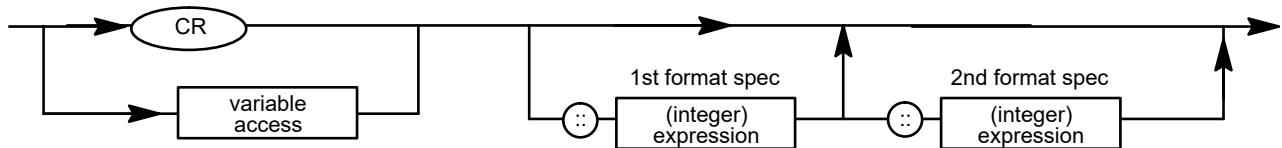


Figure E (q) OPEN FILE, OPEN HAND, PAUSE, PULSE, PURGE Statements

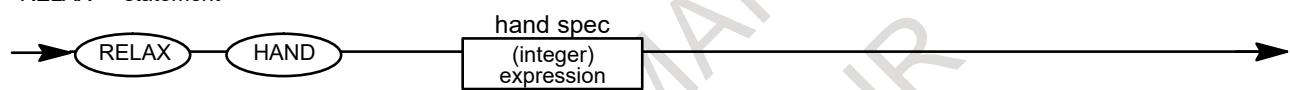
READ -- statement



read item -- statement



RELAX -- statement



RELEASE -- statement



REPEAT -- statement



RESUME -- statement

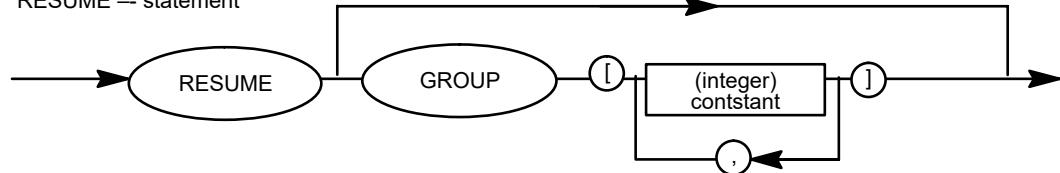
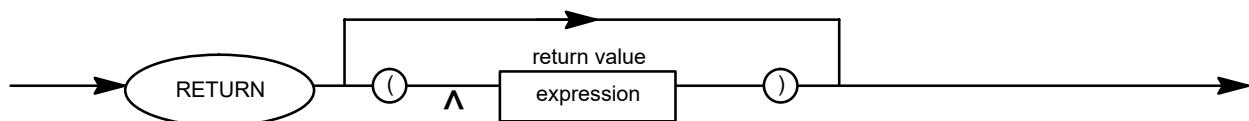
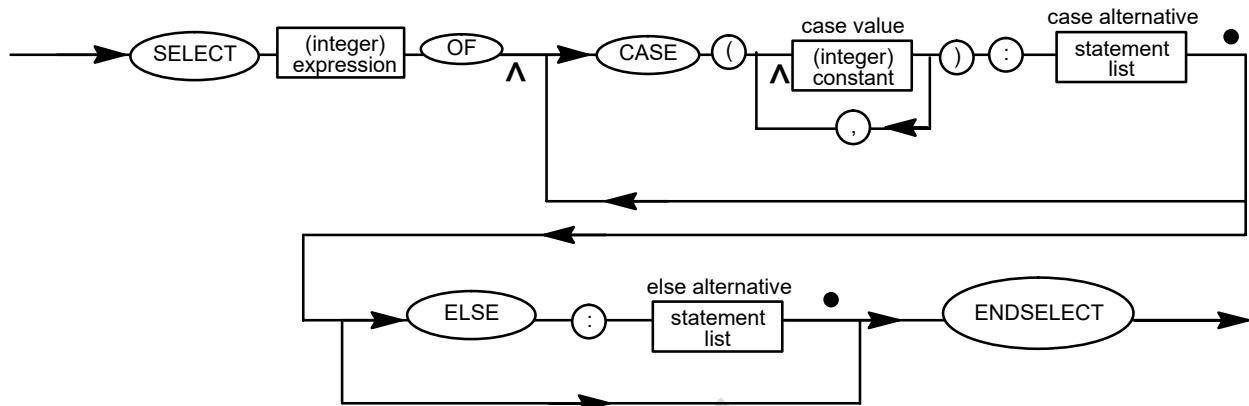


Figure E (r) READ, RELAX, RELEASE, REPEAT, RESUME Statements

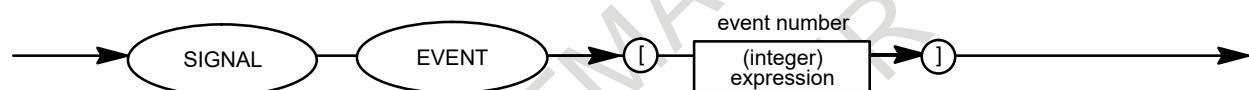
RETURN -- statement



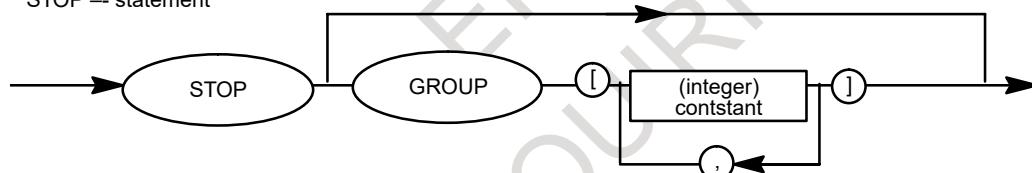
SELECT -- statement



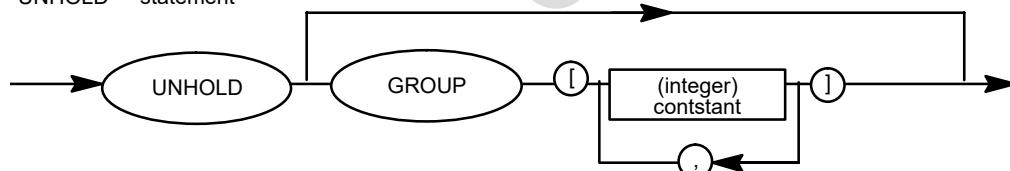
SIGNAL -- statement



STOP -- statement



UNHOLD -- statement



USING -- statement

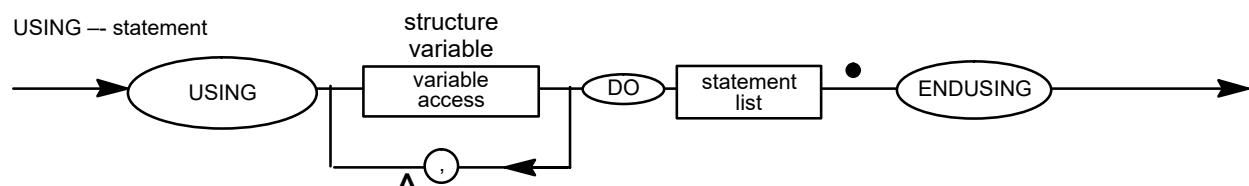
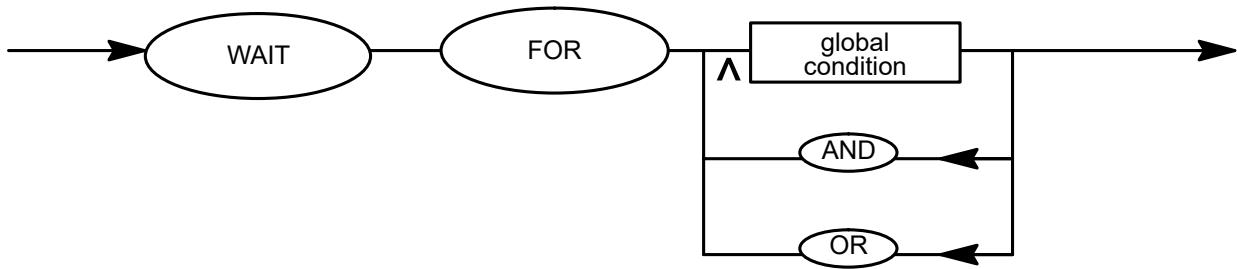


Figure E (s) RETURN, SELECT, SIGNAL, STOP, UNHOLD, USING Statements

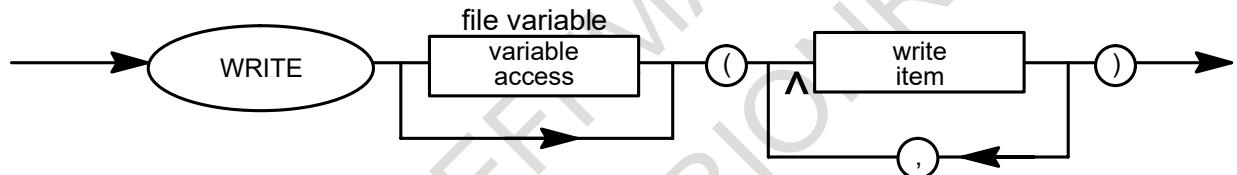
WAIT -- statement



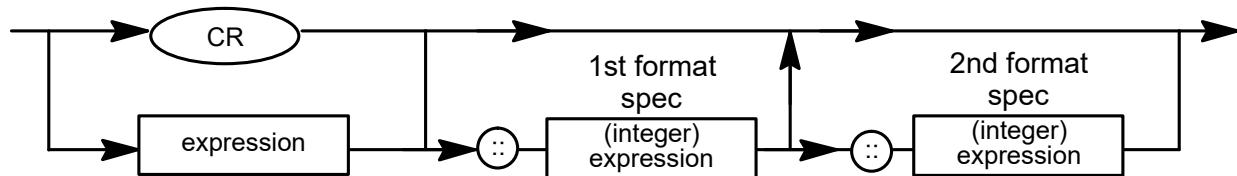
WHILE -- statement



WRITE -- statement



write item

**Figure E (t) WAIT, WHILE, WRITE Statements**

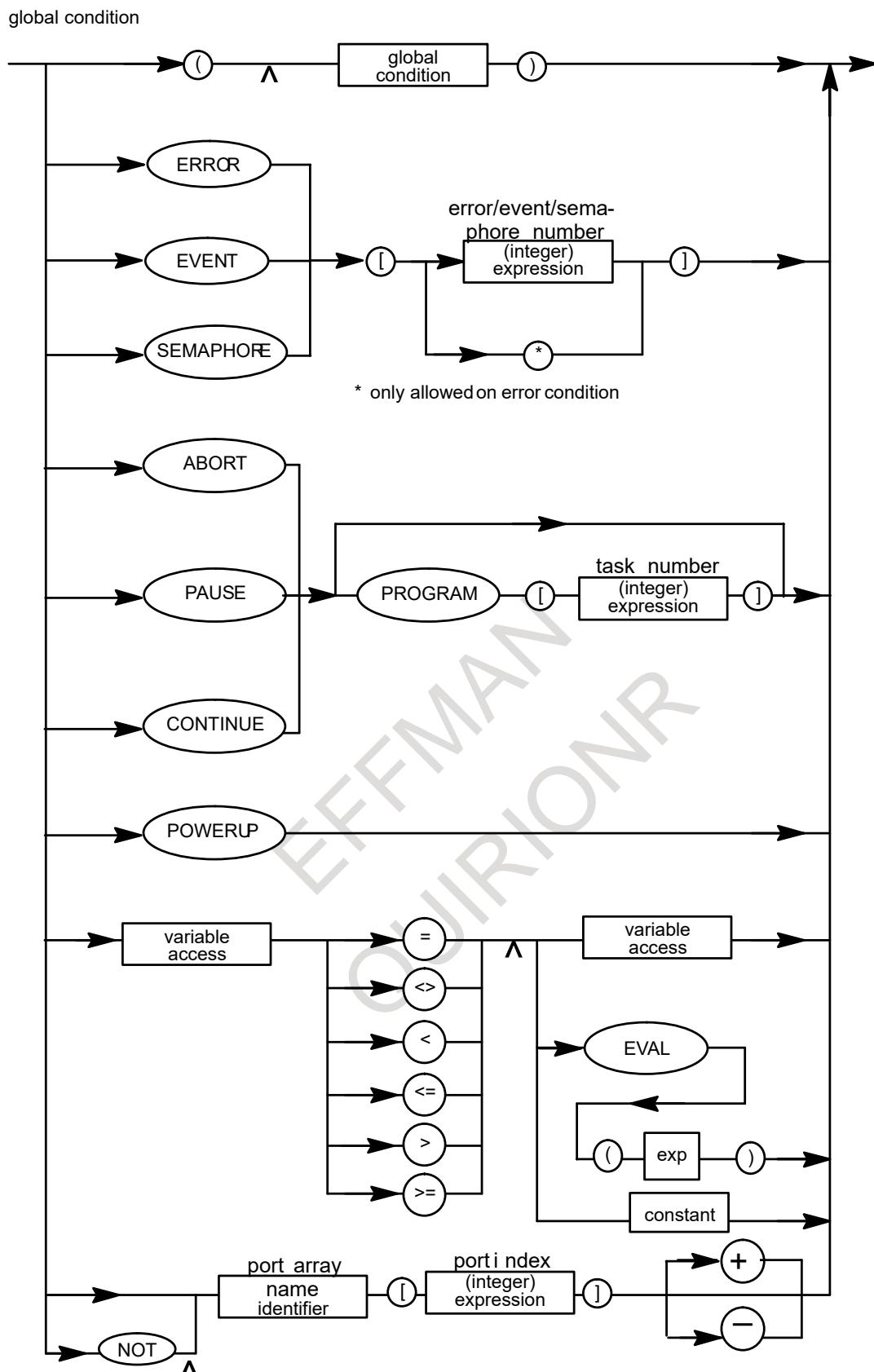
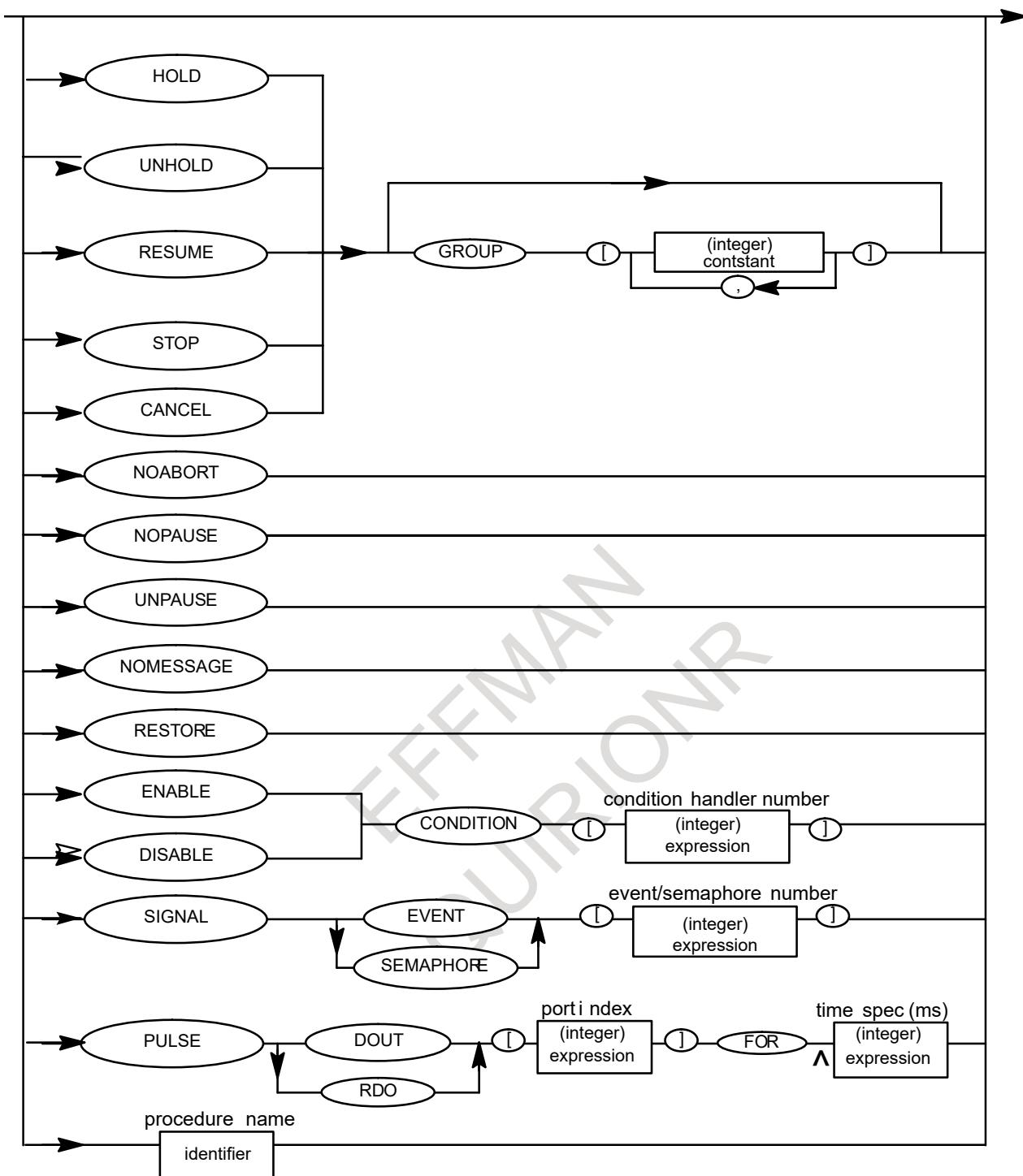


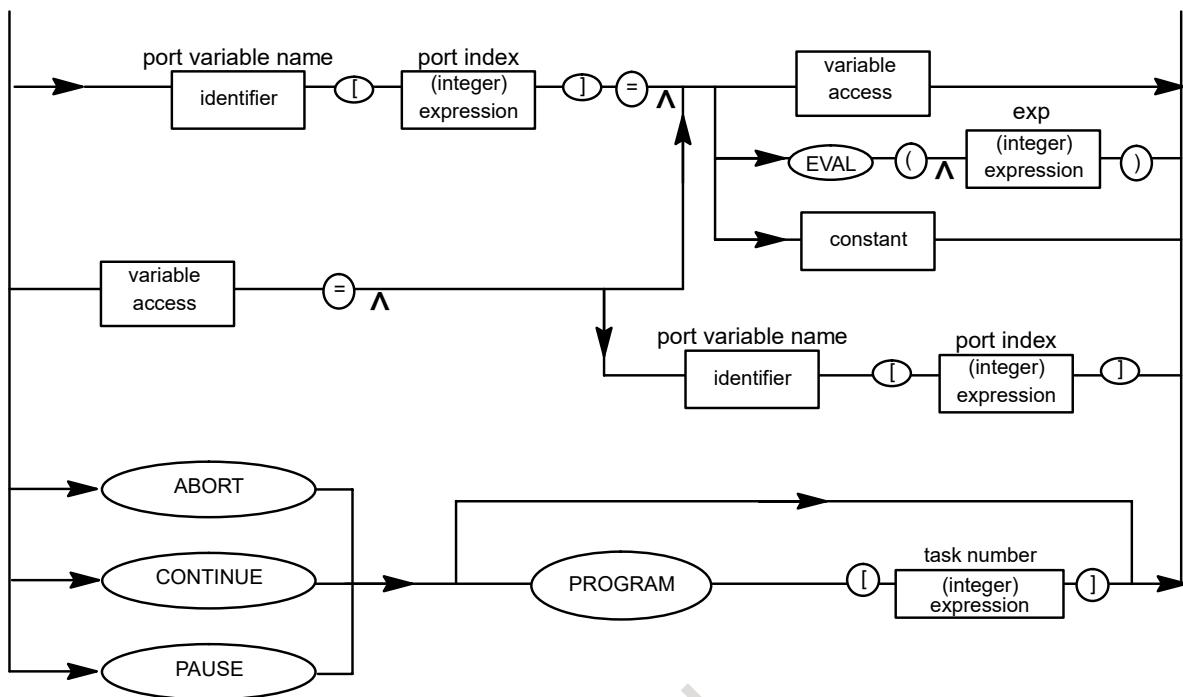
Figure E (u) Global Condition

### condition handler action



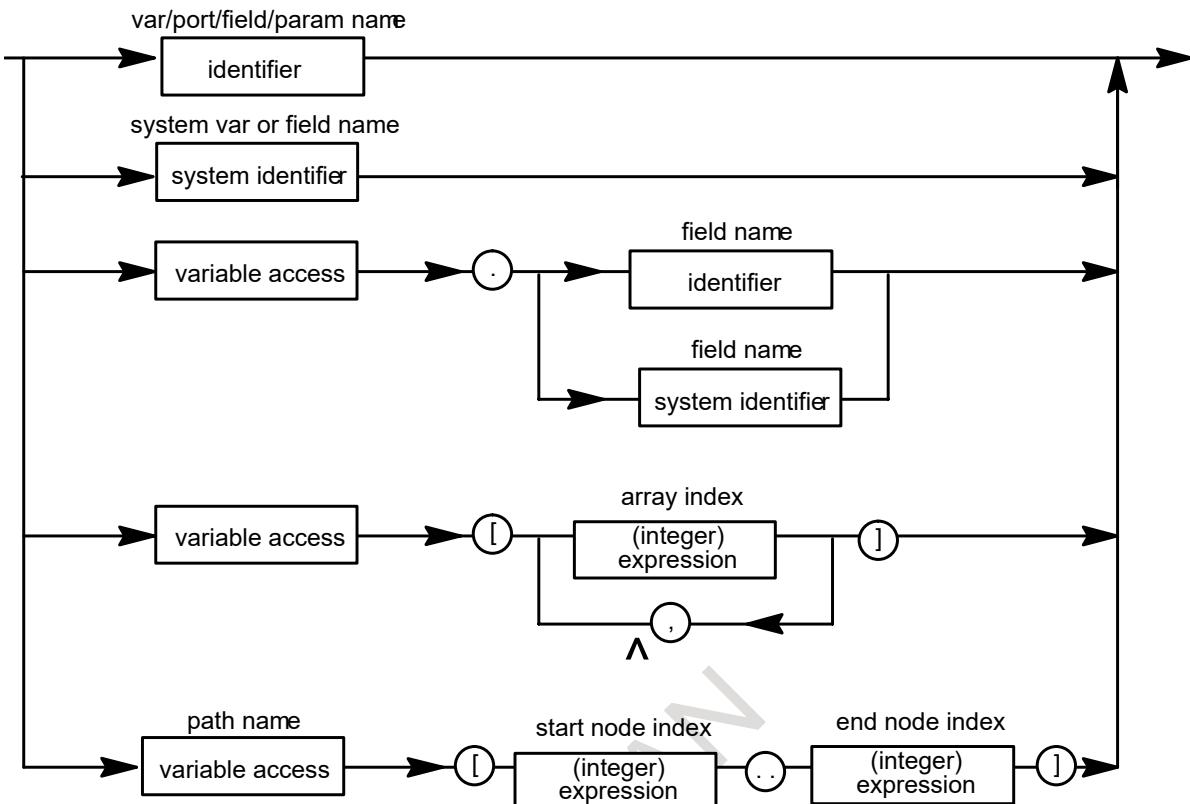
## **Figure E (v) Condition Handler Action**

condition handler action continued

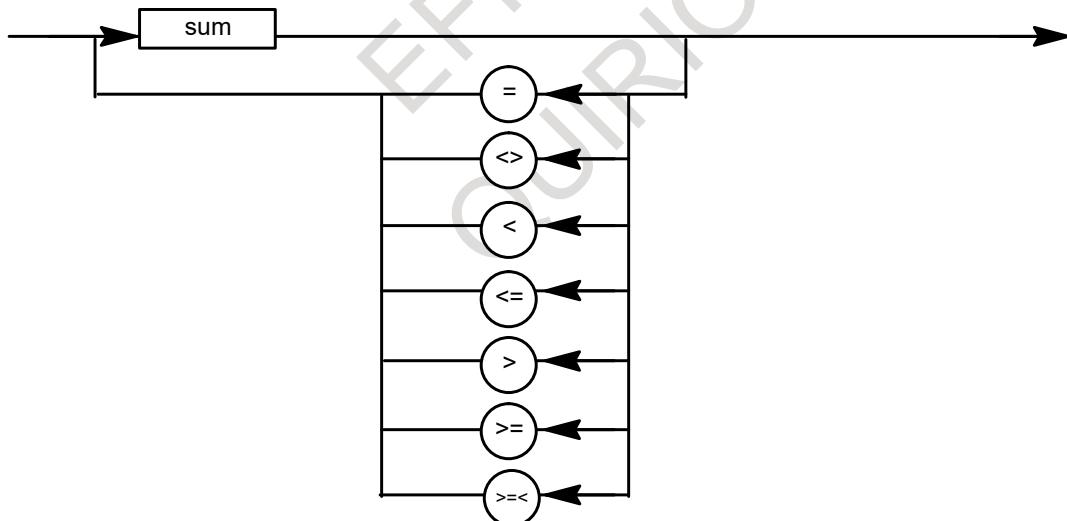


**Figure E (w) Condition Handler Action, continued**

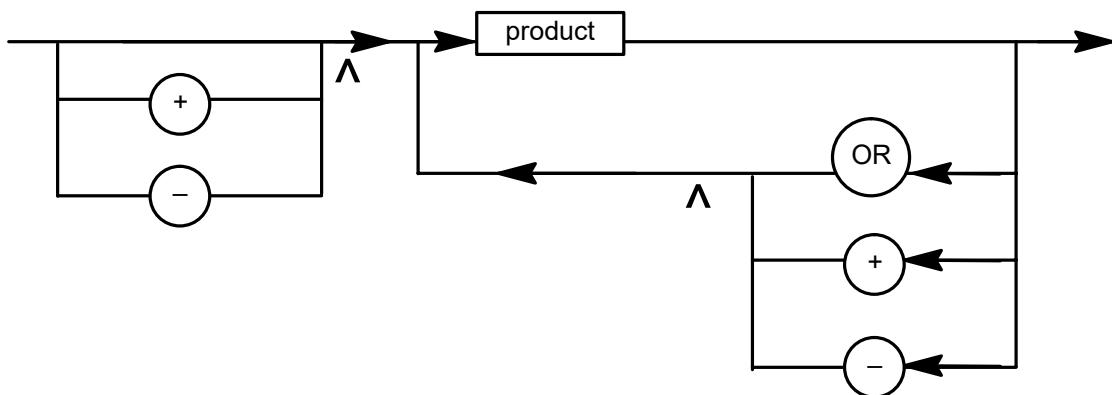
## variable access



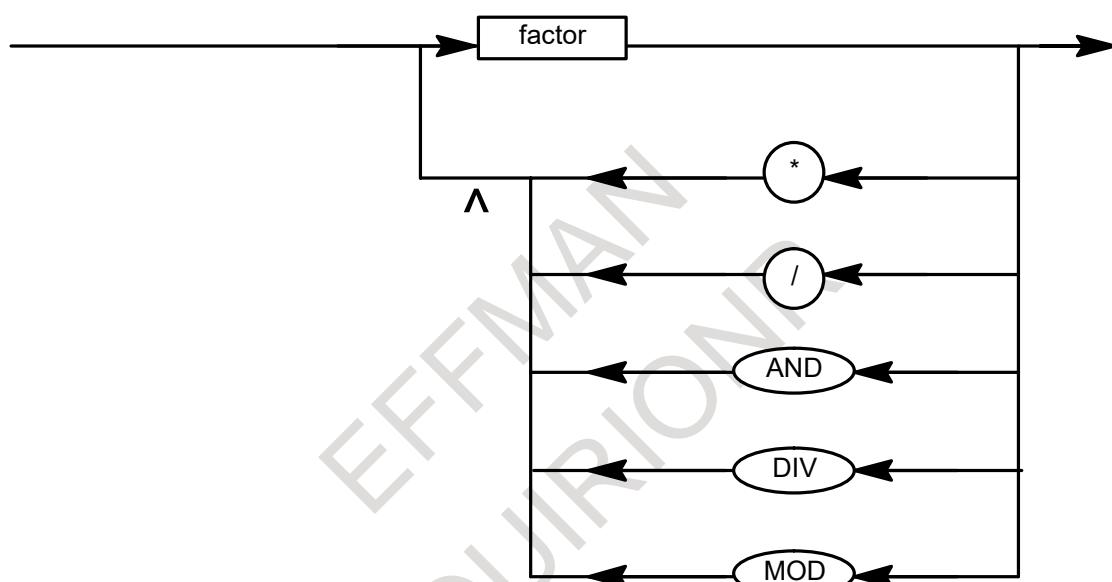
## expression

**Figure E (x) Variable Access, Expression**

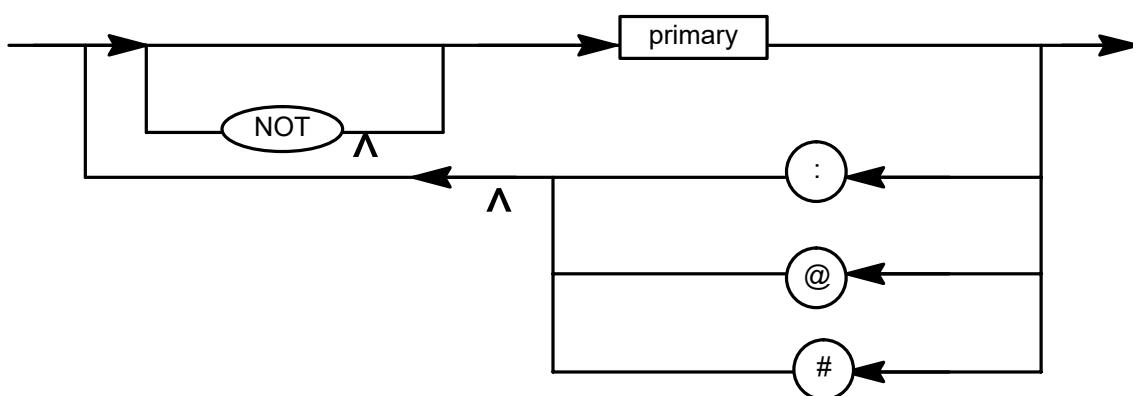
sum



## product

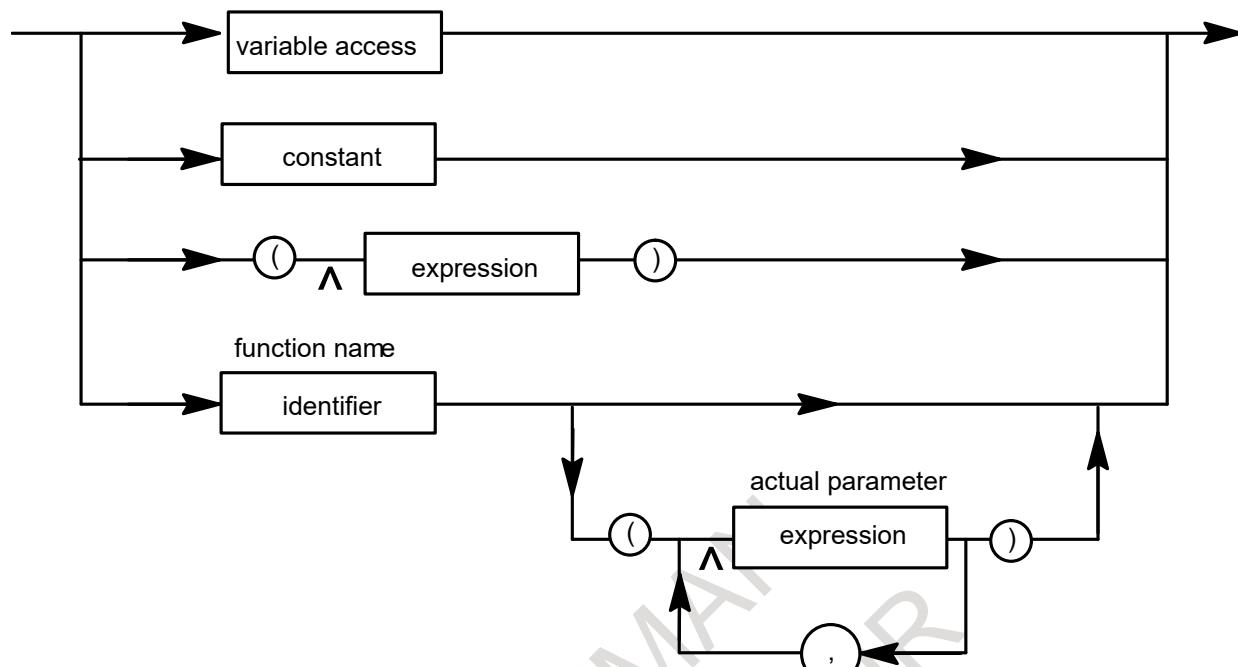


factor

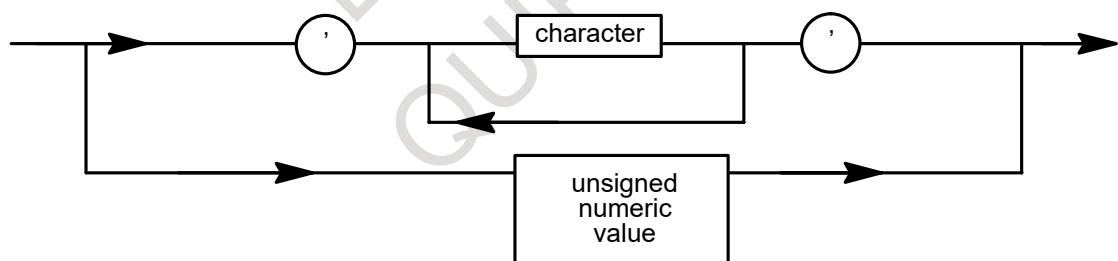


### Figure E (y) Sum, Product, Factor

primary



literal



**Figure E (z) Primary, Literal**

# F TRANSLATING BY ROBOGUIDE

This appendix shows how to create or translate KAREL programs using ROBOGUIDE. Refer to ROBOGUIDE help for details on how to use ROBOGUIDE.

Start ROBOGUIDE and newly create or start the virtual robot.

## F.1 TRANSLATE KAREL PROGRAMS

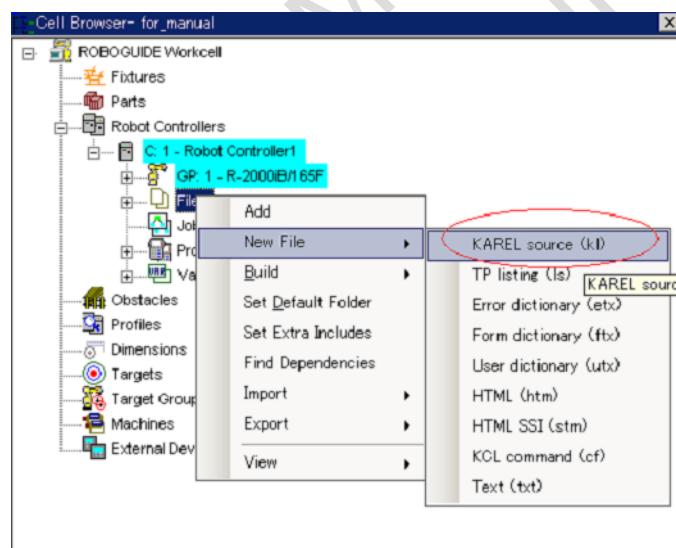
This section shows examples of translating newly created or added KAREL programs.

### F.1.1 Sample of Newly Adding a KAREL Program

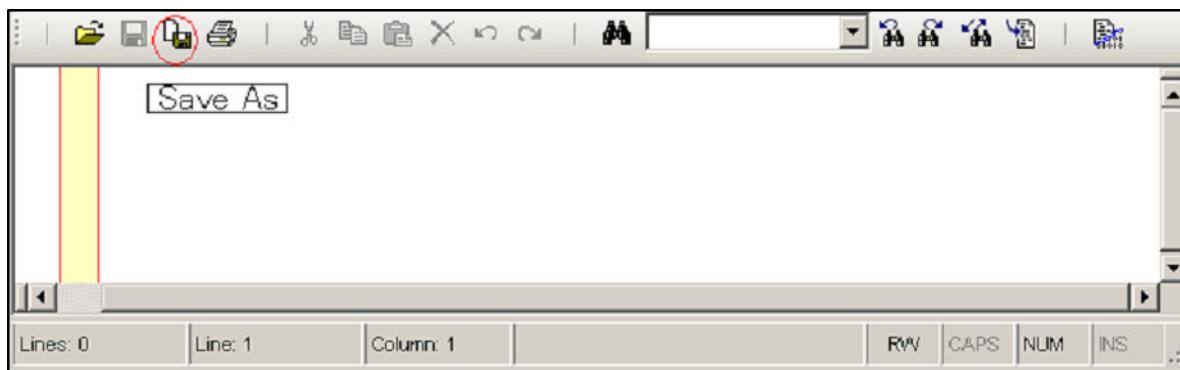
Right-click on **Cell Browser**, and then select **New File > KAREL source (kl)**. To add a created KAREL program, select **Add**.

#### NOTE

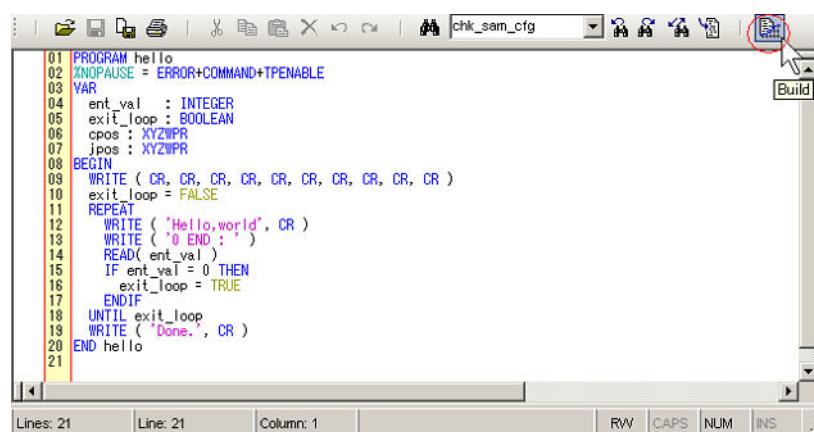
If **Cell Browser** is not displayed, select it from the menu.



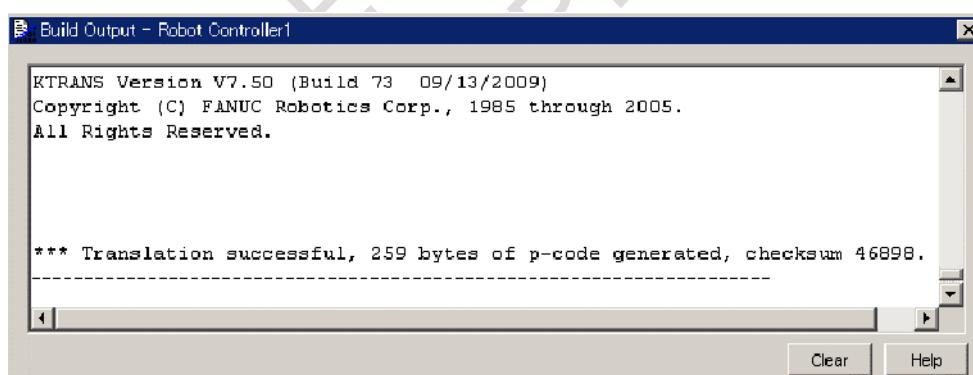
A newly added file is empty. It is named similar to Untitled1.kl. Click the **Save As** button and input the KAREL program name within eight characters, such as formtest.kl.



After inputting the KAREL program, click the upper-right **Build** button, then translate the KAREL program. The translated KAREL program is loaded to the virtual robot automatically if it has been successfully built.



The result of the build is displayed in the other window. If errors are found, a simple explanation is displayed.

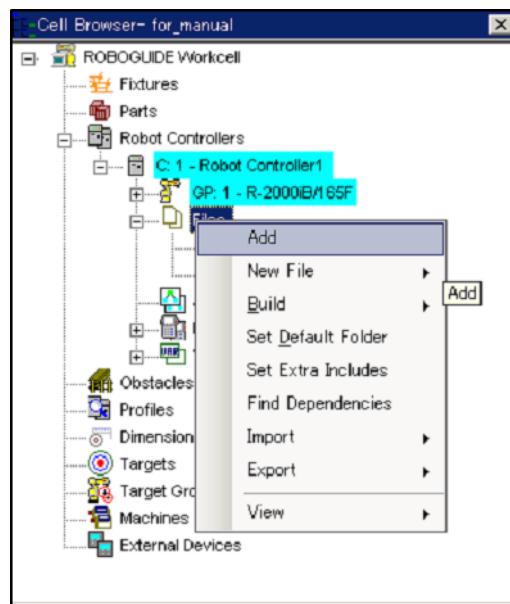


## F.1.2 Sample of Adding an Existing KAREL Program

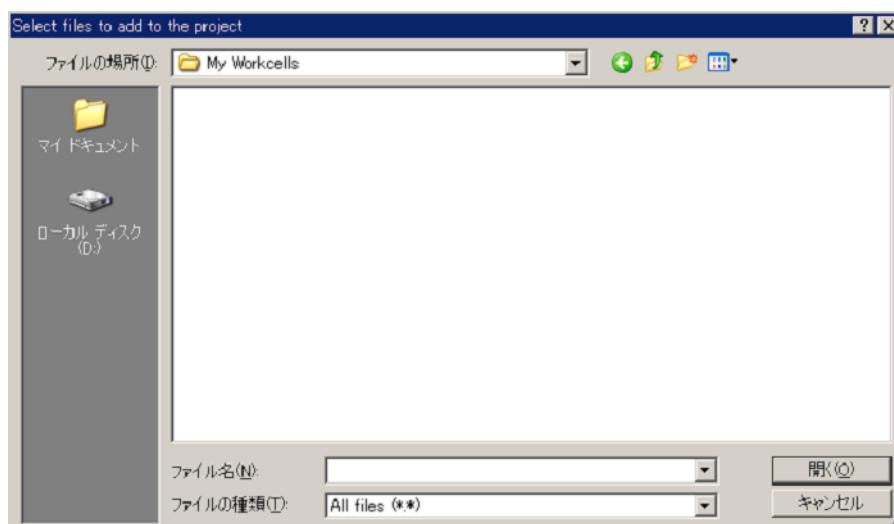
Right-click on the file in the **Cell Browser** menu and select **Add**. The following is a sample of the display.

### NOTE

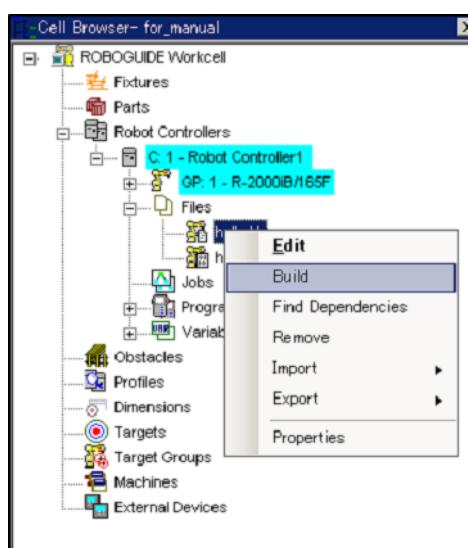
If **Cell Browser** is not displayed, select it from the menu.



The following screen is displayed. Select the KAREL program to add.



Right-click on the added file and select **Build**. Then the KAREL program is translated. You can translate dictionary files in a way similar to translating KAREL programs.



A KAREL program can be built in the same way as in [Section F.1.1, Sample of Newly Adding a KAREL Program.](#)

## F.2 TRANSLATE A DICTIONARY

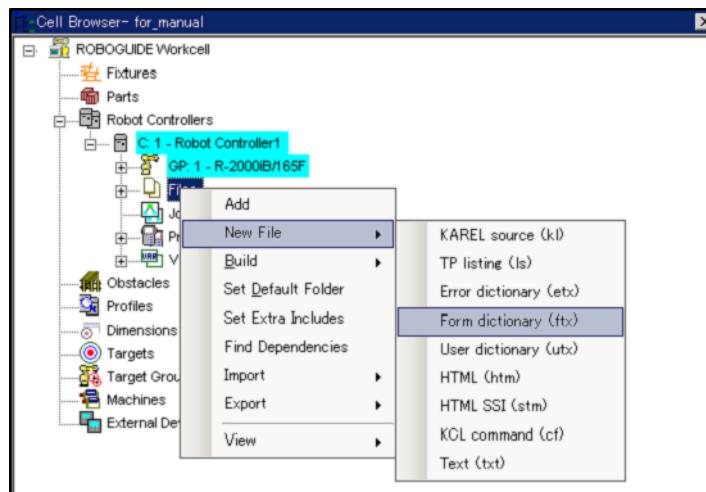
You can translate dictionary files in a way similar to translating KAREL programs. Refer to ROBOGUIDE help for information on creating dictionaries using ROBOGUIDE.

### F.2.1 Sample of Translating Newly Created Dictionaries

Right-click on **Cell Browser** and select **New File**. The following is a sample.

#### NOTE

If **Cell Browser** is not displayed, select it from the menu.

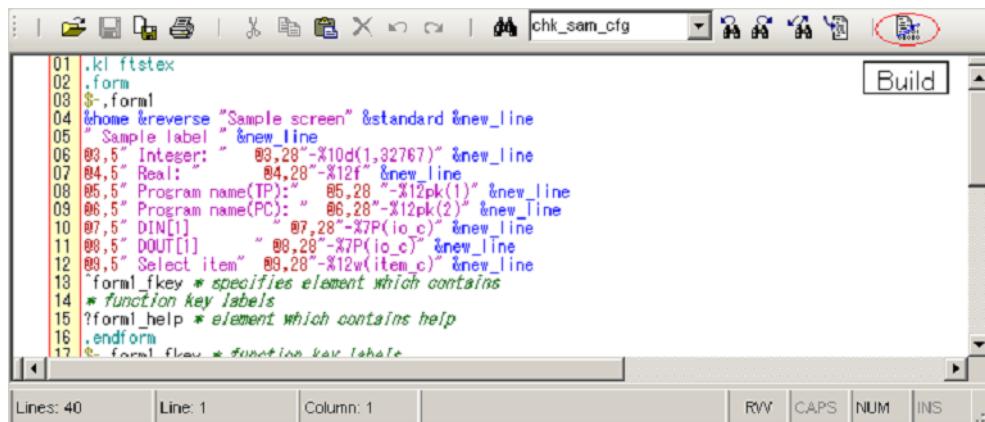


A newly added file is empty. It is named similar to `Untitled1.ftx`. Click the **Save As** button and input a dictionary name according to the rule that the first four characters are the dictionary name and the next four characters are the language name. For example, for a dictionary name of `dictengl.ftx`, `dict` is the dictionary name and `engl` is the language name.



Three files, `fstengl.ftx`, `fstjapa.ftx`, and `fstkanj.ftx` are prepared here for a robot of multiple languages. Each dictionary is English, Japanese (KANA: for the legacy pendant), and Japanese (KANJI: for the iPendant). `fstengl.ftx` is for English.

Create a dictionary and click the upper-right **Build** button.



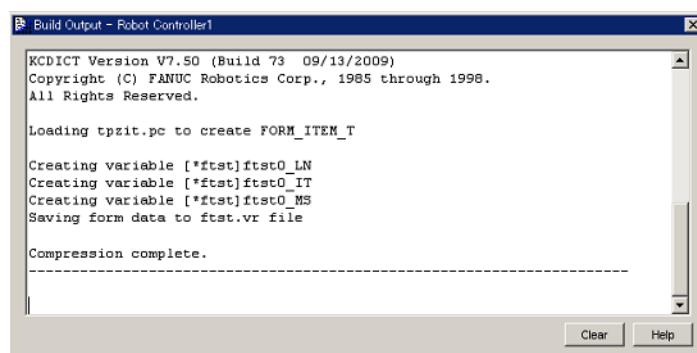
```

01 .kl ftstex
02 .form
03 $.,form1
04 &home &reverse "Sample screen" &standard &new_line
05 " Sample Label " &new_line
06 #0,5" Integer: " #03,28"-#10d(1,82767)" &new_line
07 #04,28"-#12f" &new_line
08 #06,5" Program name(TP): " #05,28"-#12pk(1)" &new_line
09 #06,5" Program name(PC): " #06,28"-#12pk(2)" &new_line
10 #07,5" DIN[1] " #07,28"-#7P(io_c)" &new_line
11 #00,5" DOUT[] " #00,28"-#7P(io_c)" &new_line
12 #00,5" Select item" #00,28"-#12w(item_c)" &new_line
13 "form1_fkey * specifies element which contains
14 * function key labels
15 ?form1_help * element which contains help
16 .endform
17 <- form1 flown * function key labels

```

Lines: 40 | Line: 1 | Column: 1 | RW | CAPS | NUM | INS

The result of the build is displayed in the other window. If errors are found, a simple explanation is displayed.



```

KCDICT Version V7.50 (Build 73 09/13/2009)
Copyright (C) FANUC Robotics Corp., 1985 through 1998.
All Rights Reserved.

Loading tpzit.pc to create FORM_ITEM_T

Creating variable [*ftst]ftst0_LN
Creating variable [*ftst]ftst0_IT
Creating variable [*ftst]ftst0_MS
Saving form data to ftst.vr file

Compression complete.
-----
```

Clear | Help

If the dictionary has been successfully translated, it is automatically loaded on the virtual robot.

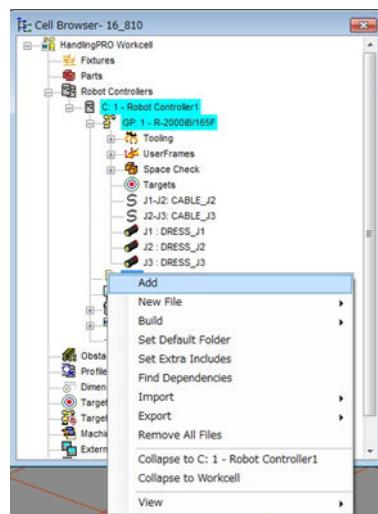
## F.2.2 Sample of Translating Added Existing Dictionaries

Select **Add** when adding a dictionary that has already been created. Right-click on the file in **Cell Browser** and select **Add**.

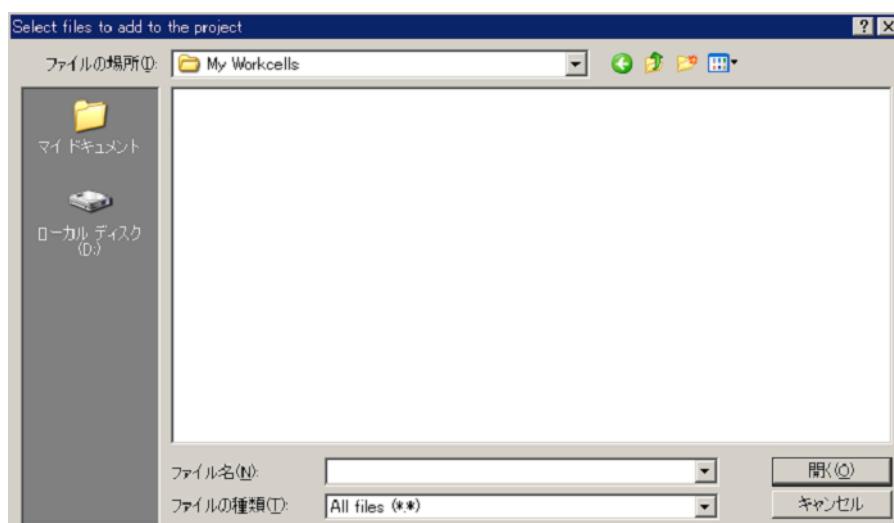
**NOTE**

If **Cell Browser** is not displayed, select it from the menu.

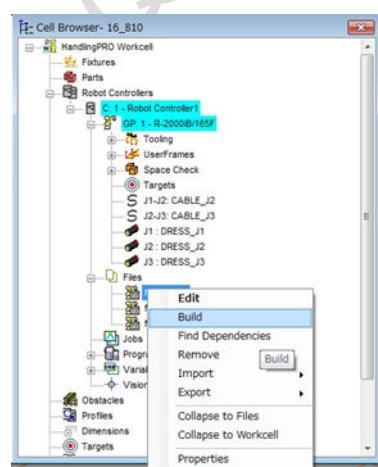
The following is a sample.



The **Select** menu is displayed. Select the dictionary to add.



Right-click on the added dictionary file on the **Cell Browser** menu, select **Build**, and then the dictionary is built.



## F.2.3 Dictionary Name and Languages

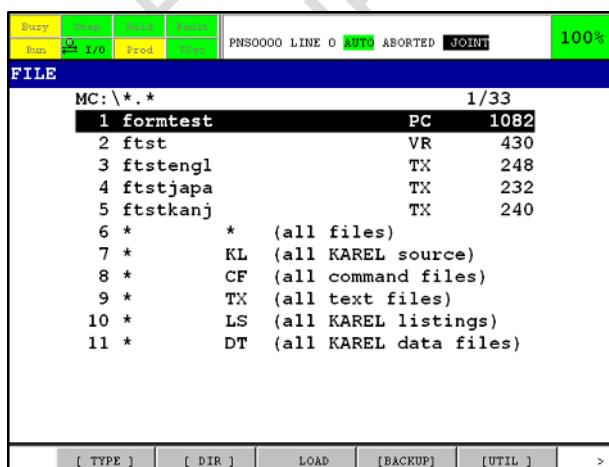
A dictionary name has the following rules:

- File name is 8 characters
- First 4 characters are the dictionary name
- Last 4 characters are the language name
- The extension is one of etx, ftx, or utx
- ftx is used for the Form Editor
- The following are the language name 4 characters:
  - ENGL = ENGLISH
  - JAPA = JAPANESE
  - KANJ = KANJI
  - FREN = FRENCH
  - GERM = GERMAN
  - SPAN = SPANISH
  - CHIN = CHINESE
  - TAIW = TAIWANESE

## F.3 LOAD ON THE ROBOT CONTROLLER

After confirming on the virtual robot, the next step is to confirm on the actual robot.

1. Copy PC, VR, and TX file to the Compact FLASH ATA card (called MC: hereinafter).
2. Insert the MC: on the robot controller.
3. Display the FILE screen. It allows PC, VR, and TX files to be loaded with F3, LOAD.



### NOTE

Only FTST.VR is used because the VR file includes the starting position and width of data. A VR file is created for each language but enabled data is only in one of them. Then you must unify kind, position, and width of data in each language. Only one VR file is loaded. Usually, we recommend the use of the English VR file.

EFFMAN  
QUIRIONR

# INDEX

---



---

## Special Characters

%ALPHABETIZE Translator Directive 260  
 %CMOS2SHADOW Translator Directive 283  
 %CMOSVARS Translator Directive 283  
 %COMMENT Translator Directive 293  
 %CRTDEVICE Translator Directive 307  
 %DEFGROUP Translator Directive 316  
 %DELAY Translator Directive 318  
 %ENVIRONMENT Translator Directive 337  
 %INCLUDE Translator Directive 376  
 %LOCKGROUP Translator Directive 405  
 %NOABORT Translator Directive 413  
 %NOBUSYLAMP Translator Directive 413  
 %NOLOCKGROUP Translator Directive 415  
 %NOPAUSE Translator Directive 416  
 %NOPAUSESHFT Translator Directive 417  
 %PRIORITY Translator Directive 434  
 %SHADOWVARS Translator Directive 508  
 %STACKSIZE Translator Directive 511  
 %TIMESLICE Translator Directive 515  
 %TPMOTION Translator Directive 516  
 %UNINITVARS Translator Directive 519  
*iRVision* 522, 524–530, 534–541, 543–545, 552–554, 556, 557, 559–562, 567–569  
*iRVISION CUSTOM SCREEN USING V CSAPI BUILT-INS* 651

## A

A KAREL Client Application 181  
 A KAREL Server Application 183  
 Abbreviations 205  
 ABORT Action 250  
 ABORT command 655  
 ABORT Condition 251  
 ABORT Statement 251  
 ABORT\_TASK Built-In Procedure 252  
 ABS Built-In Function 252  
 ACCESS RIGHTS 201  
 Accessing Dictionary Elements from a KAREL Program 145  
 ACOS Built-In Function 252  
 ACT\_SCREEN Built-In Procedure 253  
 ACT\_TBL Built-In Procedure 254  
 ACTIONS 62  
 ADD\_BYNAMEPC Built-In Procedure 255  
 ADD\_DICT Built-In Procedure 256  
 ADD\_INTPC Built-In Procedure 257  
 ADD\_REALPC Built-In Procedure 258  
 ADD\_STRINGPC Built-In Procedure 259  
 AIN and AOUT Signals 210  
 Alternation Control Structures 37  
 ANSI C Loopback Client Example 184  
 APPEND FILE command 656  
 APPEND NODE command 656  
 APPEND\_NODE Built-In Procedure 260  
 APPEND\_QUEUE Built-In Procedure 261  
 APPLYING OFFSETS TO A COPIED TEACH PENDANT PROGRAM 644

APPROACH Built-In Function 261  
 Arithmetic Operations 30  
 ARRAY Data Type 262  
 ARRAY\_LEN Built-In Function 263  
 ARRAYS 23  
 ASIN Built-In Function 263  
 Assignment Action 264  
 Assignment Actions 62  
 Assignment Statement 265  
 ATAN2 Built-In Function 266  
 ATT\_WINDOW\_D Built-In Procedure 267  
 ATT\_WINDOW\_S Built-In Procedure 268  
 ATTACH Statement 267  
 AVL\_POS\_NUM Built-In Procedure 269

## B

BOOLEAN Data Type 269  
 Boolean Operations 32  
 BYNAME Built-In Function 270  
 BYTE Data Type 271  
 BYTES\_AHEAD Built-In Procedure 271  
 BYTES\_LEFT Built-In Function 272

## C

CALL\_PROG Built-In Procedure 274  
 CALL\_PROGLIN Built-In Procedure 274  
 CANCEL Action 275  
 CANCEL FILE Statement 276  
 CANCEL Statement 275  
 Character Codes 143  
 CHARACTER CODES 701, 709  
 Character Set 11  
 CHDIR command 657  
 CHECK\_DICT Built-In Procedure 277  
 CHECK\_EPOS Built-In Procedure 277  
 CHECK\_NAME Built-In Procedure 278  
 Child Tasks 233  
 CHR Built-In Function 278  
 CLEAR ALL command 657  
 CLEAR BREAK CONDITION command 658  
 CLEAR Built-In Procedure 279  
 CLEAR\_DICT command 658  
 CLEAR\_PROGRAM command 659  
 CLEAR\_VARS command 659  
 CLEAR\_SEMA Built-In Procedure 279  
 CLOSE FILE Statement 280  
 CLOSE FILE STATEMENT 76  
 CLOSE HAND Statement 280  
 CLOSE\_TPE Built-In Procedure 281  
 CLR\_IO\_STAT Built-In Procedure 281  
 CLR\_PORT\_SIM Built-In Procedure 282  
 CLR\_POS\_REG Built-In Procedure 282  
 CNC\_DYN\_DISB Built-In Procedure 283  
 CNC\_DYN\_DISE Built-In Procedure 284  
 CNC\_DYN\_DISI Built-In Procedure 284  
 CNC\_DYN\_DISP Built-In Procedure 285  
 CNC\_DYN\_DISR Built-In Procedure 285

CNC\_DYN\_DISS Built-In Procedure 286  
 CNCL\_STP\_MTN Built-In Procedure 286  
 CNV\_CNF\_STRG Built-In Procedure 287  
 CNV\_CONF\_STR Built-In Procedure 288  
 CNV\_INT\_STR Built-In Procedure 288  
 CNV\_JPOS\_REL Built-In Procedure 289  
 CNV\_REAL\_STR Built-In Procedure 289  
 CNV\_REL\_JPOS Built-In Procedure 290  
 CNV\_STR\_CONF Built-In Procedure 291  
 CNV\_STR\_INT Built-In Procedure 291  
 CNV\_STR\_REAL Built-In Procedure 292  
 CNV\_STR\_TIME Built-In Procedure 292  
 CNV\_TIME\_STR Built-In Procedure 293  
 Command Procedure Format 205  
 COMMAND PROCEDURES 205  
 Comments 17  
 COMPARE\_FILE Built-in Procedure 294  
 COMPRESS DICT command 660  
 Compressing and Loading Dictionaries on the Controller 144  
 Compressing and Loading Forms on the Controller 163  
**CONDITION HANDLER OPERATIONS 56**  
 Condition Handlers 38  
**CONDITION HANDLERS 55**  
**CONDITION...ENDCONDITION Statement 296**  
**CONDITIONS 58**  
 CONFIG Data Type 297  
 CONFIGURING THE SOCKET MESSAGING OPTION 173  
 CONFIGURING THE SOCKET MESSAGING OPTION Overview 173  
 CONNECT TIMER Statement 298  
 CONT\_TASK Built-In Procedure 299  
 CONTINUE Action 298  
 CONTINUE command 660  
 CONTINUE Condition 299  
 CONTROLLER 6  
 COPY FILE command 661  
 COPY\_FILE Built-In Procedure 300  
 COPY\_PATH Built-In Procedure 301  
 COPY\_QUEUE Built-In Procedure 302  
 COPY\_TPE Built-In Procedure 303  
 COPYING PATH VARIABLES 594  
 COS Built-In Function 304  
 CR Input/Output Item 304  
 CREATE VARIABLE command 661  
 CREATE\_TPE Built-In Procedure 305  
 CREATE\_VAR Built-In Procedure 306  
 Creating a Program 2  
 Creating Command Procedures 206  
**CREATING USER DICTIONARIES 137**  
 CREATING USER FORMS 145  
 CRT/KB Form Screen 162  
 CURJPOS Built-In Function 308  
 CURPOS Built-In Function 308  
 CURR\_PROG Built-In Function 309  
 Cursor Position Attributes 157

**D**

DAQ\_CHECKP Built-In Procedure 310  
 DAQ\_REGPIPE Built-In Procedure 310  
 DAQ\_START Built-In Procedure 312  
 DAQ\_STOP Built-In Procedure 313  
 DAQ\_UNREG Built-In Procedure 314  
 DAQ\_WRITE Built-In Procedure 315  
**DATA TRANSFER BETWEEN ROBOTS OVERVIEW 187**

DATA TRANSFER BETWEEN ROBOTS SETUP 187  
**DATA TRANSFER BETWEEN ROBOTS TERMINOLOGY 187**  
 DATA TYPES 19  
 DEF\_SCREEN Built-In Procedure 316  
 DEF\_WINDOW Built-In Procedure 317  
 Default Program 203  
 DEL\_INST\_TPE Built-In Procedure 321  
 DELAY Statement 318  
 DELETE FILE command 662  
 DELETE NODE command 663  
 DELETE VARIABLE command 663  
 DELETE\_FILE Built-In Procedure 319  
 DELETE\_NODE Built-In Procedure 320  
 DELETE\_QUEUE Built-In Procedure 320  
 DET\_WINDOW Built-In Procedure 321  
 Device Name 114  
**DICTIONARIES AND FORMS 137**  
**DICTIONARIES AND FORMS OVERVIEW 137**  
 Dictionary Comment 143  
 Dictionary Cursor Positioning 139  
 Dictionary Element Name 138  
 Dictionary Element Number 138  
 Dictionary Element Text 139  
 Dictionary Name and Languages 741  
 Dictionary Reserved Word Commands 141  
 Dictionary Syntax 137  
 DIN and DOUT Signals 209  
 DIRECTORY command 663  
 DISABLE BREAK PROGRAM command 664  
 DISABLE CONDITION Action 322  
 DISABLE CONDITION Statement 322  
 DISCONNECT TIMER Statement 323  
 DISCTRL\_ALPH Built-In Procedure 324  
 DISCTRL\_FORM Built-In Procedure 325  
 DISCTRL\_LIST Built-In Procedure 327  
 DISCTRL\_PLMN Built-In Procedure 328  
 DISCTRL\_SBMN Built-In Procedure 329  
 DISCTRL\_TBL Built-In Procedure 331  
 DISMOUNT command 665  
 DISMOUNT\_DEV Built-In Procedure 333  
 DISP\_DAT\_T Data Type 333  
 Display Only Data Items 157  
 Displaying a Form 164  
**DISPLAYING A LIST FROM A DICTIONARY FILE 632**  
 DOSFILE\_INF  
     built-in procedure 335  
 DOSFILE\_INF Built-In Procedure 335  
 dtbr  
     data transfer between robots 187

**E**

EDIT command 665  
 Edit Data Item 151, 157  
 ENABLE BREAK PROGRAM 665  
 ENABLE CONDITION Action 336  
 ENABLE CONDITION command 666  
 ENABLE CONDITION Statement 336  
 ENTERING COMMANDS 204  
 ERR\_DATA Built-In Procedure 338  
 ERROR Condition 339  
 Error Messages 205  
 Error Processing 206  
 ERROR RECOVERY 195

- Ethernet Device 117  
 EVAL Clause 340  
 Evaluation of Expressions and Assignments 27  
**EVENT Condition 340**  
 Example KAREL Program Referencing an XML File 129  
 Exchanging Data during a Socket Messaging Connection 180  
 Executing a Program 4  
 Executing Command Procedures 206  
 Execution Control Statements 38  
 Execution History 4  
 EXP Built-In Function 341  
**EXPRESSIONS AND ASSIGNMENTS 27**
- F**
- F-ROM Disk 113  
**FILE ACCESS 128**  
**FILE Data Type 341**  
**FILE INPUT/OUTPUT OPERATIONS 67**  
**FILE INPUT/OUTPUT OPERATIONS OVERVIEW 67**  
 File Name 115  
 File Pipes 124  
**FILE SPECIFICATION 113**  
 File String 73  
**FILE SYSTEM 113**  
**FILE SYSTEM OVERVIEW 113**  
 File Type 116  
**FILE VARIABLES 67**  
**FILE\_LIST Built-In Procedure 342**  
 filtered memory device 117  
 Flash File Storage disk (FR) 117  
 FMD device 117  
**FOR...ENDFOR Statement 343**  
**FORCE\_LINK Built-In Procedure 344**  
**FORCE\_SPMENU Built-In Procedure 345**  
 Form Attributes 148  
 Form File Naming Convention 162  
 Form Function Key Element Name or Number 159  
 Form Function Key Using a Variable 160  
 Form Help Element Name or Number 161  
 Form Menu Text 149  
 Form Reserved Words and Character Codes 158  
 Form Selectable Menu Item 150  
 Form Syntax 146  
 Form Title and Menu Label 148  
**FORMAT command 666**  
**FORMAT\_DEV Built-In Procedure 347**  
**FORMATTING BINARY INPUT/OUTPUT 88**  
 Formatting BOOLEAN Data Items 83, 90  
 Formatting INTEGER Data Items 80, 89  
 Formatting JOINTPOS Data Items 92  
 Formatting POSITION Data Items 91  
 Formatting Positional Data Items 87  
 Formatting REAL Data Items 82, 90  
 Formatting STRING Data Items 85, 90  
**FORMATTING TEXT (ASCII) INPUT/OUTPUT 79**  
 Formatting VECTOR Data Items 87, 91  
**FORMATTING XML INPUT 128**  
 Formatting XML Input Overview 128  
 Formatting XYZWPR Data Items 91  
 Formatting XYZWPREXT Data Items 92  
**FRAME Built-In Function 348**  
 Frames 9  
**FRAMES OF REFERENCE 96**  
**FROM Clause 349**  
 From KAREL Programs 234  
 From KCL 234  
 From TPP Programs 233
- G**
- Generating a KAREL Constant File 143  
**GET\_ATTR\_PRG Built-In Procedure 349**  
**GET\_FILE\_POS Built-In Function 351**  
**GET\_JPOS\_REG Built-In Function 352**  
**GET\_JPOS\_TPE Built-In Function 352**  
**GET\_PORT\_ASG Built-in Procedure 353**  
**GET\_PORT\_ATR Built-In Function 354**  
**GET\_PORT\_CMT Built-In Procedure 356**  
**GET\_PORT\_MOD Built-In Procedure 356**  
**GET\_PORT\_SIM Built-In Procedure 358**  
**GET\_PORT\_VAL Built-In Procedure 358**  
**GET\_POS\_FRM Built-In Procedure 359**  
**GET\_POS\_REG Built-In Function 359**  
**GET\_POS\_TPE Built-In Function 360**  
**GET\_POS\_TYP Built-In Procedure 361**  
**GET\_PREG\_CMT Built-In-Procedure 361**  
**GET\_QUEUE Built-In Procedure 362**  
**GET\_REG Built-In Procedure 363**  
**GET\_REG\_CMT Built-In Procedure 363**  
**GET\_SREG\_CMT Built-In Procedure 364**  
**GET\_STR\_REG Built-In Procedure 364**  
**GET\_TIM Built-In Procedure 365**  
**GET\_TPE\_CMT Built-in Procedure 365**  
**GET\_TPE\_PRM Built-in Procedure 366**  
**GET\_TSK\_INFO Built-In Procedure 368**  
**GET\_USEC\_SUB Built-In Procedure 369**  
**GET\_USEC\_TIM Built-In Function 369**  
**GET\_VAR Built-In Procedure 370**  
 GIN and GOUT Signals 210  
 Global Condition Handlers 56  
 GO TO Statement 373  
 Guidelines for a Good Implementation 180
- H**
- Hand Signals 211  
**HELP command 666**  
**HOLD Action 374**  
**HOLD command 667**  
**HOLD Statement 374**
- I**
- IF ... ENDIF Statement 375  
**IN Clause 376**  
**IN\_RANGE Built-In Function 393**  
**INDEX Built-In Function 377**  
**INI\_DYN\_DISB Built-In Procedure 378**  
**INI\_DYN\_DISE Built-In Procedure 379**  
**INI\_DYN\_DISI Built-In Procedure 380**  
**INI\_DYN\_DISP Built-In Procedure 381**  
**INI\_DYN\_DISR Built-In Procedure 382**  
**INI\_DYN\_DISS Built-In Procedure 383**  
**INIT\_QUEUE Built-In Procedure 384**  
**INIT\_TBL Built-In Procedure 384**  
**INPUT/OUTPUT BUFFER 78**  
 Input/Output System 8  
**INPUT/OUTPUT SYSTEM 209**  
**INSERT NODE command 667**

INSERT\_NODE Built-In Procedure 393  
 INSERT\_QUEUE Built-In Procedure 394  
 Installation Sequence 129  
 INTEGER Data Type 395  
 INTERPRETER ASSIGNMENT 230  
 INV Built-In Function 396  
 IO\_MOD\_TYPE Built-In Procedure 396  
 IO\_STATUS Built-In Function 397

**J**

J\_IN\_RANGE Built-In Function 398  
 J740  
     order number 187  
 JOG COORDINATE SYSTEMS 98  
 JOINT2POS Built-In Function 399  
 JOINTPOS Data Type 399

**K**

KAREL BUILT-INS 196  
 KAREL COMMAND LANGUAGE (KCL) 203  
 KAREL LANGUAGE OVERVIEW 1  
 KAREL Paths 104  
 KAREL Positions 101  
 KAREL PROGRAMMING LANGUAGE 1  
 KAREL Programming Language Overview 1  
 KAREL Variables 111  
 KCL Built-In Procedure 400  
 KCL COMMAND ALPHABETICAL DESCRIPTION 655  
 KCL COMMAND FORMAT 203  
 KCL\_NO\_WAIT Built-In Procedure 401  
 KCL\_STATUS Built-In Procedure 402

**L**

Labels 15  
 LANGUAGE COMPONENTS 11  
 LANGUAGE ELEMENTS 11  
 LIMITATIONS 197  
 LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS 614  
 LN Built-In Function 402  
 LOAD ALL command 668  
 LOAD Built-In Procedure 403  
 LOAD DICT command 668  
 LOAD FORM command 669  
 LOAD MASTER command 669  
 LOAD ON THE ROBOT CONTROLLER 741  
 LOAD PROGRAM command 669  
 LOAD SERVO command 670  
 LOAD SYSTEM command 670  
 LOAD TP command 671  
 LOAD VARS command 672  
 LOAD\_STATUS Built-In Procedure 404  
 Loading Program Logic and Data 3  
 LOCK\_GROUP Built-In Procedure 404  
 LOGOUT command 673  
 Looping Control Statements 37

**M**

MANIPULATING VALUES OF DYNAMICALLY DISPLAYED VARIABLES 630

Memory 6  
 memory card 113  
 memory card (MC) 117  
 memory device  
     filtered 117  
 MEMORY DEVICE 133  
 memory device binary 117  
 Memory File Devices 122  
 memory stick)  
     USB 113  
 MIRROR Built-In Function 406  
 Miscellaneous Actions 64  
 MKDIR command 673  
 MODIFY\_QUEUE Built-In Procedure 407  
 Modifying KAREL Variables 111  
 MOTION CONTROL 230  
 Motion Related Actions 63  
 MOTION\_CTL Built-In Function 408  
 MOUNT command 673  
 MOUNT\_DEV Built-In Procedure 409  
 MOVE FILE command 674  
 MOVE\_FILE Built-In Procedure 409  
 MSG\_CONNECT Built-In Procedure 410  
 MSG\_CONNECT(string, integer) 179  
 MSG\_DISCO Built-In Procedure 411  
 MSG\_DISCO(string, integer) 179  
 MSG\_PING Built-In Procedure 412  
 MSG\_PING(string, integer) 179  
 Multi-Dimensional Arrays 24  
 MULTI-TASKING 229  
 MULTI-TASKING TERMINOLOGY 229

**N**

NETWORK PERFORMANCE 180  
 NETWORK PERFORMANCE OVERVIEW 180  
 NOABORT Action 412  
 NODE\_SIZE Built-In Function 413  
 NOMESSAGE Action 415  
 Non>Selectable Text 157  
 NOPAUSE Action 416

**O**

OPEN FILE Statement 417  
 OPEN FILE STATEMENT 68  
 OPEN HAND Statement 418  
 OPEN\_TPE Built-In Procedure 418  
 OPERATIONS 29  
 Operator Panel Input and Output Signals (OPIN/OPOUT) 212  
 Operators 13  
 ORD Built-In Function 419  
 order number  
     J740 187  
 ORIENT Built-In Function 420

**P**

Parse Errors 133  
 PATH Data Type 420  
 PATH\_LEN Built-In Function 422  
 PAUSE Action 422  
 PAUSE command 674  
 PAUSE Condition 423  
 PAUSE Statement 423

PAUSE\_TASK Built-In Procedure 424  
 PEND\_SEMA Built-In Procedure 425  
 personal computer 113  
 PIPE\_CONFIG Built-In Procedure 426  
 POP\_KEY\_RD Built-In Procedure 426  
 Port\_Id Action 427  
 Port\_Id Condition 427  
 Port\_Id Conditions 59  
 POS Built-In Function 428  
 POS\_REG\_TYPE Built-In Procedure 430  
 POS2JOINT Built-In Function 429  
 POSITION DATA 95  
 POSITION DATA SET AND CONDITION HANDLERS  
 PROGRAM 610  
 POSITION Data Type 431, 432  
 POSITIONAL DATA 95  
 POST\_ERR\_L Built-In Procedure 432  
 POST\_SEMA Built-In Procedure 433  
 Predefined Identifiers 15  
 PRINT command 675  
 PRINT\_FILE Built-In Procedure 434  
 Priority Scheduling 231  
 PROG\_BACKUP Built-In Procedure 435  
 PROG\_CLEAR Built-In Procedure 436  
 PROG\_LIST Built-In Procedure 438  
 PROG\_RESTORE Built-In Procedure 439  
 PROGRAM CONTROL COMMANDS 204  
 PROGRAM CONTROL OVERVIEW 37  
 PROGRAM CONTROL STRUCTURES 37  
 PROGRAM Statement 440  
 Program Structure 4  
 PULSE Action 441  
 PULSE Statement 441  
 PURGE command 675  
 PURGE CONDITION Statement 442  
 PURGE\_DEV Built-In Procedure 443  
 PUSH\_KEY\_RD Built-In Procedure 443

## Q

QUEUE\_ATTACH Built-in Procedure 444  
 QUEUE\_TYPE Data Type 446

## R

RAM Disk 113  
 RAM Disk (RD) 117  
 READ Statement 446  
 READ STATEMENT 76  
 READ\_DICT Built-In Procedure 447  
 READ\_DICT\_V Built-In-Procedure 448  
 READ\_KB Built-In Procedure 449  
 REAL Data Type 452  
 RECORD command 676  
 Relational Condition 453  
 Relational Conditions 59  
 Relational Operations 31  
 RELAX HAND Statement 454  
 RELEASE Statement 455  
 REMOVE\_DICT Built-In Procedure 455  
 RENAME FILE command 676  
 RENAME VARS command 677  
 RENAME\_FILE Built-In Procedure 456  
 RENAME\_VAR Built-In Procedure 456  
 RENAME\_VARS Built-In Procedure 457

REPEAT ... UNTIL Statement 457  
 Reserved Words 13  
 RESET Built-In Procedure 458  
 RESET command 678  
 RESUME Action 458  
 RESUME Statement 459  
 RETURN Statement 460  
 RGET\_PORTCMT Built-In Routine 460  
 RGET\_PORTSIM Built-In Routine 461  
 RGET\_PORTVAL Built-In Routine 462  
 RGET\_PREGCMT Built-In Routine 462  
 RGET\_REG Built-In Routine 463  
 RGET\_REG\_CMT Built-In Routine 464  
 RGET\_SREGCMT Built-in Routine 465  
 RGET\_STR\_REG Built-In Routine 465  
 RGETPREG: Program to Get Position Register 192  
 RMCN\_ALERT Built-In Routine 466  
 RMCN\_SEND Built-in Routine 467  
 RMDIR command 678  
 RNUMREG\_RECV Built-In Routine 468  
 RNUMREG\_SEND Built-In Routine 469  
 Robot Digital Input and Output Signals (RDI/RDO) 212  
 ROUND Built-In Function 470  
 Routine Call Actions 64  
 ROUTINE Statement 470  
 ROUTINES 39  
 RPREG\_RECV Built-In Routine 471  
 RPREG\_SEND Built-in Routine 472  
 RSET\_INT\_REG Built-in Routine 473  
 RSET\_PORTCMT Built-in Routine 474  
 RSET\_PORTSIM Built-in Routine 475  
 RSET\_PORTVAL Built-in Routine 475  
 RSET\_PREGCMT Built-in Routine 476  
 RSET\_REALREG Built-in Routine 477  
 RSET\_REG\_CMT Built-In Routine 477  
 RSET\_SREGCMT Built-in Routine 478  
 RSET\_STR\_REG Built-in Routine 479  
 RSETNREG: Program to Set Numeric Register 192  
 RSETPREG: Program to Set Position Register 194  
 Rule for Expressions and Assignments 27  
 RUN command 678  
 RUN\_TASK Built-In Procedure 479  
 RUNCF command 679  
 Running Programs from the User Operator Panel (UOP) PNS  
 Signal 233

## S

Sample of Adding an Existing KAREL Program 736  
 Sample of Newly Adding a KAREL Program 735  
 Sample of Translating Added Existing Dictionaries 739  
 Sample of Translating Newly Created Dictionaries 738  
 SAVE Built-In Procedure 481  
 SAVE MASTER command 679  
 SAVE SERVO command 680  
 SAVE SYSTEM command 680  
 SAVE TP command 680  
 SAVE VARS command 681  
 SAVE\_DRAM Built-In Procedure 481  
 SAVING DATA TO THE DEFAULT DEVICE 602  
 SELECT ... ENDSELECT Statement 482  
 SELECT\_TPE Built-In Procedure 482, 483  
 SEMAPHORE Condition 483  
 SEND\_DATAPC Built-In Procedure 484  
 SEND\_EVENTPC Built-In Procedure 485

SERIAL INPUT/OUTPUT 225  
SET BREAK CONDITION command 682  
SET BREAK PROGRAM command 682  
SET CLOCK command 683  
SET DEFAULT command 683  
SET GROUP command 684  
SET LANGUAGE command 684  
SET LOCAL VARIABLE command 684  
SET PORT command 685  
SET TASK command 685  
SET TRACE command 686  
SET VARIABLE command 686  
SET VERIFY command 687  
SET\_ATTR\_PRG Built-In Procedure 485  
SET\_CURSOR Built-In Procedure 486  
SET\_EPOS\_REG Built-In Procedure 487  
SET\_EPOS\_TPE Built-In Procedure 488  
SET\_FILE\_ATR Built-In Procedure 489  
SET\_FILE\_POS Built-In Procedure 489  
SET\_INT\_REG Built-In Procedure 490  
SET\_JPOS\_REG Built-In Procedure 491  
SET\_JPOS\_TPE Built-In Procedure 491  
SET\_LANG Built-In Procedure 492  
SET\_PERCH Built-In Procedure 493  
SET\_PORT\_ASG Built-In Procedure 493  
SET\_PORT\_ATR Built-In Function 494  
SET\_PORT\_CMT Built-In Procedure 496  
SET\_PORT\_MOD Built-In Procedure 497  
SET\_PORT\_SIM Built-In Procedure 497  
SET\_PORT\_VAL Built-In Procedure 498  
SET\_POS\_REG Built-In Procedure 499  
SET\_POS\_TPE Built-In Procedure 500  
SET\_PREG\_CMT Built-In-Procedure 500  
SET\_REAL\_REG Built-In Procedure 501  
SET\_REG\_CMT Built-In-Procedure 501  
SET\_SREG\_CMT Built-in Procedure 501  
SET\_STR\_REG Built-in Procedure 502  
SET\_TIME Built-In Procedure 502  
SET\_TPE\_CMT Built-In Procedure 503  
SET\_TRNS\_TPE Built-In Procedure 504  
SET\_TSK\_ATTR Built-In Procedure 504  
SET\_TSK\_NAME Built-In Procedure 505  
SET\_VAR Built-In Procedure 506  
Setting File and Port Attributes 68  
Setting up a Client Tag 176  
Setting up a Server Tag 174  
SETTING UP DIGITAL OUTPUT PORTS FOR PROCESS MONITORING 585  
SHORT Data Type 508  
SHOW BREAK command 687  
SHOW BUILTINS command 688  
SHOW CLOCK command 688  
SHOW CONDITION command 688  
SHOW CURPOS command 689  
SHOW data\_type command 695  
SHOW DEFAULT command 689  
SHOW DEVICE command 689  
SHOW DICTS command 689  
SHOW GROUP command 690  
SHOW HISTORY command 690  
SHOW LANG command 690  
SHOW LANGS command 690  
SHOW LOCAL VARIABLE command 690  
SHOW LOCAL VARS command 691  
SHOW MEMORY command 692  
SHOW PROGRAM command 692  
SHOW PROGRAMS command 692  
SHOW SYSTEM command 693  
SHOW TASK command 693  
SHOW TASKS command 693  
SHOW TRACE command 694  
SHOW TYPES command 694  
SHOW VARIABLE command 694  
SHOW VARS command 695  
SIGNAL EVENT Action 509  
SIGNAL EVENT Statement 509  
SIGNAL SEMAPHORE Action 509  
SIMULATE command 696  
SIN Built-In Function 510  
SKIP command 696  
SOCKET MESSAGING 173  
SOCKET MESSAGING AND KAREL 178  
SOCKET MESSAGING AND KAREL OVERVIEW 178  
Socket Messaging Hardware Requirements 173  
SOCKET MESSAGING OVERVIEW 173  
SOCKET MESSAGING PROGRAMMING EXAMPLES 180  
Socket Messaging Software Requirements 173  
Socket Messaging System Requirements Overview 173  
Software Components 5  
Special Operations 32  
SQRT Built-In Function 510  
STANDARD DATA TRANSFER PROGRAMS 190, 191  
STANDARD ROUTINES 604  
STARTING TASKS 232  
STD\_PTH\_NODE Data Type 511  
STEP OFF command 697  
STEP ON command 697  
STOP Action 511  
STOP Statement 512  
STORAGE 201  
STORAGE DEVICE ACCESS 117  
Storage Device Access Overview 117  
STR\_LEN Built-In Function 513  
STRING Data Type 512  
STRUCTURE Data Type 514  
SUB\_STR Built-In Function 514  
Subdirectories 205  
Supported Robots 6  
System and Program Event Conditions 60  
SYSTEM REQUIREMENTS 173  
SYSTEM SOFTWARE 5  
System Variables 17  
SYSTEM VARIABLES 201  
SYSTEM-DEFINED SIGNALS 212

**T**

TAN Built-In Function 515  
TASK CONTROL AND MONITORING 233  
TASK SCHEDULING 231  
TCP/IP Setup 188  
TCP/IP SETUP FOR ROBOGUIDE 189  
Teach Pendant Form Screen 161  
Teach Pendant Input and Output Signals (TPIN/TPOUT) 219  
Teaching KAREL Paths 105  
Teaching KAREL Positions 101  
TEACHING KAREL VARIABLES 101  
TIME OUT AND RETRY 197  
Time Slicing 232  
Tool Definition (UTOOL) 97

TRANSLATE A DICTIONARY 738  
 TRANSLATE Built-In Procedure 516  
 TRANSLATE command 698  
 TRANSLATE KAREL PROGRAMS 735  
 Translating a Program 2  
 TRANSLATING BY ROBOGUIDE 735  
 TRANSLATOR DIRECTIVES 18  
 TROUBLESHOOTING 198  
 TRUNC Built-In Function 517  
 TYPE command 698

**U**

Unconditional Branch Statement 38  
 UNHOLD Action 518  
 UNHOLD Statement 518  
 UNINIT Built-In Function 519  
 UNLOCK\_GROUP Built-In Procedure 519  
 UNPAUSE Action 521  
 UNPOS Built-In Procedure 521  
 UNSIMULATE command 698  
 Usage String 74  
 USB memory stick 113  
 USB Memory Stick Device (UD1) 117  
 USB Memory Stick Device (UT1) 117  
 USE OF OPERATORS 27  
 User Frame (UFRAME) 97  
 User Interface Devices 8  
 USER INTERFACE TIPS 92  
 USER Menu on the CRT/KB 93  
 USER Menu on the Teach Pendant 92  
 User-Defined Data Structures 22  
 User-Defined Data Types 20  
 USER-DEFINED DATA TYPES AND STRUCTURES 20  
 User-Defined Identifiers 14  
 USER-DEFINED SIGNALS 209  
 USING ... ENDUSING Statement 522  
 USING DYNAMIC DISPLAY BUILT-INS 622  
 Using Frames in the Teach Pendant Editor (TP) 98  
 USING REGISTER BUILT-INS 606  
 USING SEMAPHORES AND TASK SYNCHRONIZATION 234  
 USING THE DISCTRL\_ALPHA BUILT-IN 641  
 USING THE FILE AND DEVICE BUILT-INS 619

**V**

V\_ACQ\_VAMAP iRVision Built-In Procedure 522  
 V\_ADJ\_2D iRVision Built-In Procedure 523  
 V\_CAM\_CALIB iRVision Built-In Procedure 524  
 V\_CAM\_CHECK iRVision Built-In Procedure 525  
 V\_CLR\_VAMAP iRVision Built-In Procedure 526  
 V\_CSAPI\_GETVALUE iRVision Built-In Procedure 526  
 V\_CSAPI\_NUMSET iRVision Built-In Procedure 527  
 V\_CSAPI\_RESETDATA iRVision Built-In Procedure 527  
 V\_CSAPI\_SAVEDATA iRVision Built-In Procedure 528  
 V\_CSAPI\_SETVALUE iRVision Built-In Procedure 528  
 V\_CSAPI\_TESTRUN iRVision Built-In Procedure 529  
 V\_DISPLAY4D iRVision Built-In Procedure 530  
 V\_FIND\_VIEW iRVision Built-In Procedure 530  
 V\_FIND\_VLINE iRVision Built-In Procedure 531  
 V\_GET\_FOUND iRVision Built-In Procedure 534  
 V\_GET\_OFFSET iRVision Built-In Procedure 535  
 V\_GET\_PASSFL iRVision Built-In Procedure 536  
 V\_GET\_READ iRVision Built-In Procedure 537

V\_GET\_VPARAM iRVision Built-In Procedure 538  
 V\_IRCONNECT iRVision Built-In Procedure 539  
 V\_LED\_OFF iRVision Built-In Procedure 540  
 V\_LED\_ON iRVision Built-In Procedure 540  
 V\_OVERRIDE iRVision Built-In Procedure 541  
 V\_RUN\_FIND iRVision Built-In Procedure 541  
 V\_SAVE\_IMREG iRVision Built-In Procedure 543  
 V\_SET\_REF iRVision Built-In Procedure 544  
 V\_SNAP\_VIEW iRVision Built-In Procedure 545  
 VAR\_INFO Built-In Procedure 546  
 VAR\_LIST Built-In Procedure 548  
 Variable-Sized Arrays 25  
 Variables and Data Types 203  
 Variables and Expressions 29  
 VECTOR Data Type 550  
 Virtual Devices 123  
 VOL\_SPACE Built-In Procedure 550  
 VREG\_FND\_POS iRVision Built-in Procedure 552  
 VREG\_OFFSET iRVision Built-in Procedure 552  
 VT\_ACK\_QUEUE iRVision Built-In Procedure 553  
 VT\_CLR\_QUEUE iRVision Built-In Procedure 553  
 VT\_DELETE\_PQ iRVision Built-In Procedure 554  
 VT\_GET\_AREID iRVision Built-In Procedure 556  
 VT\_GET\_FOUND iRVision Built-In Procedure 556  
 VT\_GET\_LINID iRVision Built-In Procedure 557  
 VT\_GET\_PFRT iRVision Built-In Procedure 557  
 VT\_GET\_QUEUE iRVision Built-In Procedure 559  
 VT\_GET\_TIME iRVision Built-In Procedure 560  
 VT\_GET\_TRYID iRVision Built-In Procedure 560  
 VT\_PUT\_QUEUE2 iRVision Built-In Procedure 561  
 VT\_PUT\_QUEUE iRVision Built-In Procedure 562  
 VT\_READ\_PQ iRVision Built-In Procedure 562  
 VT\_SET\_FLAG iRVision Built-In Procedure 567  
 VT\_SET\_LDBAL iRVision Built-In Procedure 568  
 VT\_WRITE\_PQ iRVision Built-In Procedure 569

**W**

WAIT command 699  
 WAIT FOR Statement 571  
 WHEN Clause 571  
 WHILE...ENDWHILE Statement 572  
 WITH Clause 572  
 World Frame 97  
 WRITE Statement 573  
 WRITE STATEMENT 78  
 WRITE\_DICT Built-In Procedure 573  
 WRITE\_DICT\_V Built-In Procedure 574

**X**

XML\_ADDTAG Built-In Procedure 575  
 XML\_GETDATA Built-In Procedure 576  
 XML\_REMTAG Built-In Procedure 577  
 XML\_SCAN Built-In Procedure 577  
 XML\_SETVAR Built-In Procedure 578  
 XYZWPR Data Type 579  
 XYZWPREXT Data Type 579

EFFMAN  
QUIRIONR

## REVISION RECORD

---

Edition	Date	Contents
N	Apr., 2023	<ul style="list-style-type: none"><li>• Revised built-in routines</li></ul>
M	Feb., 2022	<ul style="list-style-type: none"><li>• Added and revised <i>iRVision</i> built-in routines</li></ul>

EFFMAN  
QUIRIONR

EFFMAN  
QUIRIONR