FANUC Robot series

R-30*i*A/R-30*i*A Mate CONTROLLER

KAREL

OPERATOR'S MANUAL (Reference)

B-83144EN-1/01

• **Original Instructions**

Before using the Robot, be sure to read the "FANUC Robot Safety Manual (B-80687EN)" and understand the content.

SAFETY PRECAUTIONS

Thank you for purchasing FANUC Robot.
This chapter describes the precautions which must be observed to ensure the safe use of the robot.
Before attempting to use the robot, be sure to read this chapter thoroughly.

Before using the functions related to robot operation, read the relevant operator's manual to become familiar with those functions.

If any description in this chapter differs from that in the other part of this manual, the description given in this chapter shall take precedence.

For the safety of the operator and the system, follow all safety precautions when operating a robot and its peripheral devices installed in a work cell.
In addition, refer to the “FANUC Robot SAFETY HANDBOOK (B-80687EN)”.

1 WORKING PERSON

The personnel can be classified as follows.

Operator:
- Turns robot controller power ON/OFF
- Starts robot program from operator’s panel

Programmer or teaching operator:
- Operates the robot
- Teaches robot inside the safety fence

Maintenance engineer:
- Operates the robot
- Teaches robot inside the safety fence
- Maintenance (adjustment, replacement)

- An operator cannot work inside the safety fence.
- A programmer, teaching operator, and maintenance engineer can work inside the safety fence. The working activities inside the safety fence include lifting, setting, teaching, adjusting, maintenance, etc..
- To work inside the fence, the person must be trained on proper robot operation.

During the operation, programming, and maintenance of your robotic system, the programmer, teaching operator, and maintenance engineer should take additional care of their safety by using the following safety precautions.

- Use adequate clothing or uniforms during system operation
- Wear safety shoes
- Use helmet

2 DEFINITION OF WARNING, CAUTION AND NOTE

To ensure the safety of user 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 attempting to use the oscillator.

⚠WARNING
Applied when there is a danger of the user being injured or when there is a danger of both the user being injured and the equipment being damaged if the approved procedure is not observed.

⚠CAUTION
Applied when there is a danger of the equipment being damaged, if the approved procedure is not observed.

NOTE
Notes are used to indicate supplementary information other than Warnings and Cautions.

- Read this manual carefully, and store it in a sales place.

3 WORKING PERSON SAFETY

Working person safety is the primary safety consideration. Because it is very dangerous to enter the operating space of the robot during automatic operation, adequate safety precautions must be observed.
The following lists the general safety precautions. Careful consideration must be made to ensure working person safety.

(1) Have the robot system working persons attend the training courses held by FANUC.

FANUC provides various training courses. Contact our sales office for details.

(2) Even when the robot is stationary, it is possible that the robot is still in a ready to move state, and is waiting for a signal. In this state, the robot is regarded as still in motion. To ensure working person safety, provide the system with an alarm to indicate visually or aurally that the robot is in motion.
(3) Install a safety fence with a gate so that no working person can enter the work area without passing through the gate. Install an interlocking device, a safety plug, and so forth in the safety gate so that the robot is stopped as the safety gate is opened.

The controller is designed to receive this interlocking signal of the door switch. When the gate is opened and this signal received, the controller stops the robot (Please refer to "STOP TYPE OF ROBOT" in SAFETY PRECAUTIONS for detail of stop type). For connection, see Fig.3 (a) and Fig.3 (b).

(4) Provide the peripheral devices with appropriate grounding (Class A, Class B, Class C, and Class D).

(5) Try to install the peripheral devices outside the work area.
(6) Draw an outline on the floor, clearly indicating the range of the robot motion, including the tools such as a hand.
(7) Install a mat switch or photoelectric switch on the floor with an interlock to a visual or aural alarm that stops the robot when a working person enters the work area.
(8) If necessary, install a safety lock so that no one except the working person in charge can turn on the power of the robot.

The circuit breaker installed in the controller is designed to disable anyone from turning it on when it is locked with a padlock.

(9) When adjusting each peripheral device independently, be sure to turn off the power of the robot
(10) Operators should be ungloved while manipulating the operator’s panel or teach pendant. Operation with gloved fingers could cause an operation error.
(11) Programs, system variables, and other information can be saved on memory card or USB memories. Be sure to save the data periodically in case the data is lost in an accident.
(12) The robot should be transported and installed by accurately following the procedures recommended by FANUC. Wrong transportation or installation may cause the robot to fall, resulting in severe injury to workers.
(13) In the first operation of the robot after installation, the operation should be restricted to low speeds. Then, the speed should be gradually increased to check the operation of the robot.
(14) Before the robot is started, it should be checked that no one is in the area of the safety fence. At the same time, a check must be made to ensure that there is no risk of hazardous situations. If detected, such a situation should be eliminated before the operation.
(15) When the robot is used, the following precautions should be taken. Otherwise, the robot and peripheral equipment can be adversely affected, or workers can be severely injured.
- Avoid using the robot in a flammable environment.
- Avoid using the robot in an explosive environment.
- Avoid using the robot in an environment full of radiation.
- Avoid using the robot under water or at high humidity.
- Avoid using the robot to carry a person or animal.
- Avoid using the robot as a stepladder. (Never climb up on or hang from the robot.)
(16) When connecting the peripheral devices related to stop(safety fence etc.) and each signal (external emergency , fence etc.) of robot. be sure to confirm the stop movement and do not take the wrong connection.
(17) When preparing trestle, please consider security for installation and maintenance work in high place according to Fig.3 (c). Please consider footstep and safety bolt mounting position.

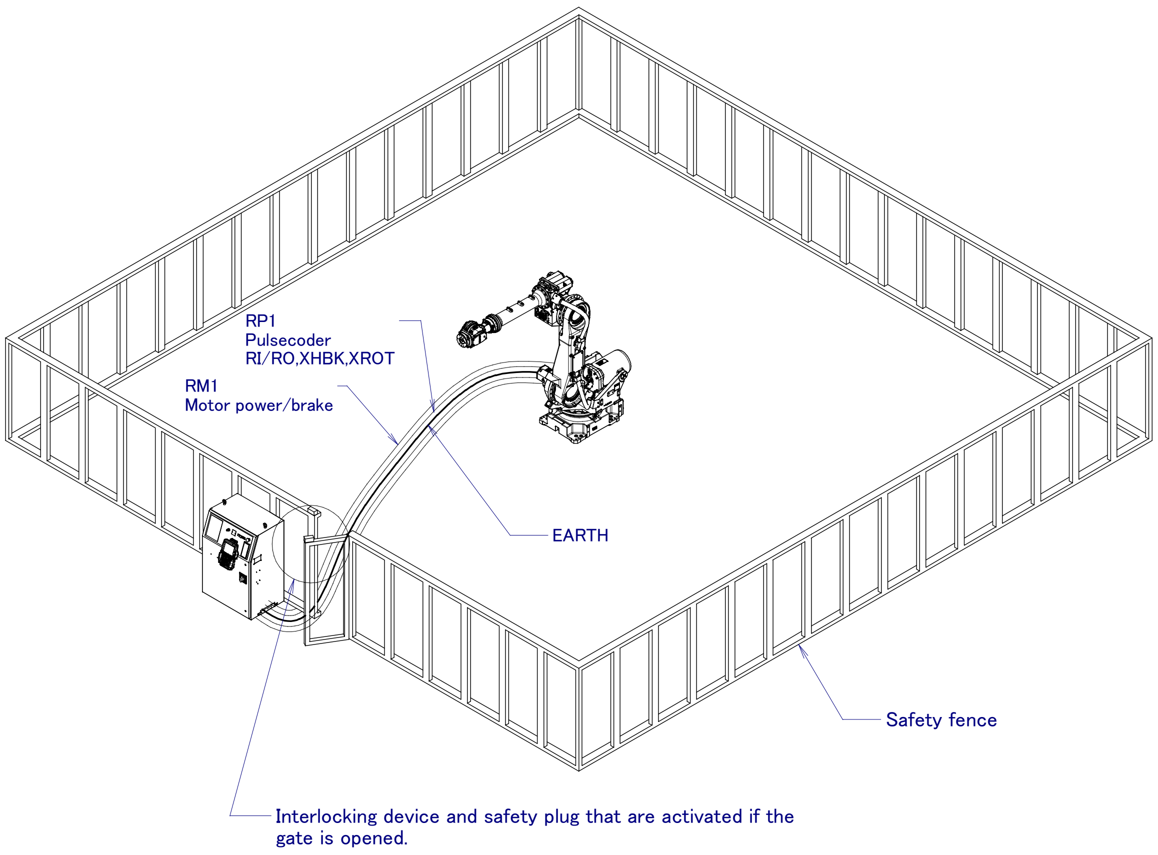
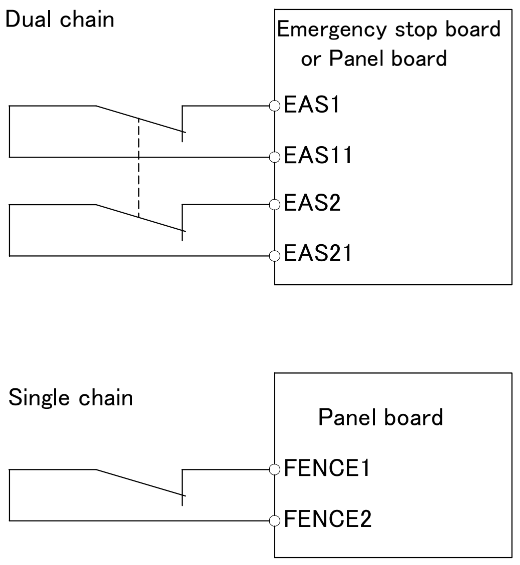

RP1
Pulsecoder
RI/RO,XHBK,XROT
RM1
Motor power/brake
EARTH
Safety fence
Interlocking device and safety plug that are activated if the gate is opened.

Fig. 3 (a) Safety fence and safety gate



(Note)

In case R-30*i*A
Terminals EAS1,EAS11,EAS2,EAS21 are provided on the emergency stop board or connector panel

In case R-30*i*A Mate
Terminals EAS1,EAS11,EAS2,EAS21 or FENCE1,FENCE2 are provided on the emergency stop board or in the connector panel of CRM65 (Open air type).

Refer to the ELECTRICAL CONNCETIONS Chapter of CONNECTION of controller maintenance manual for details.

Fig. 3 (b) Limit switch circuit diagram of the safety fence

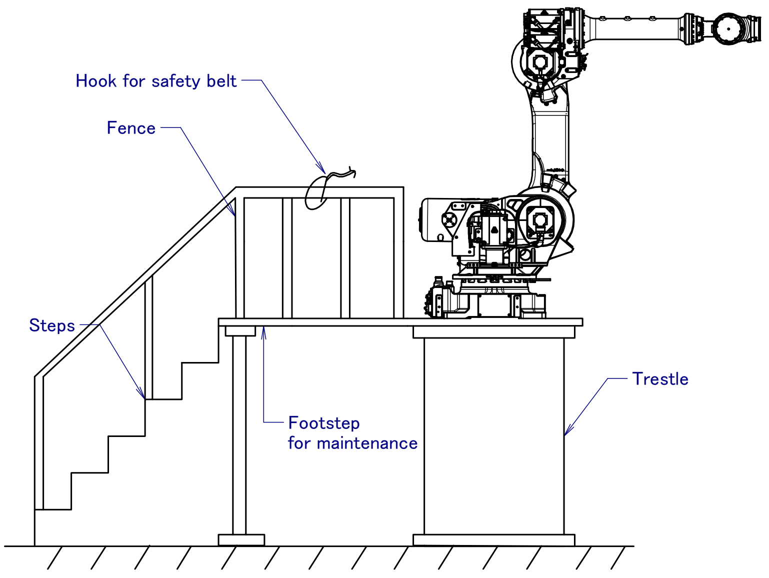


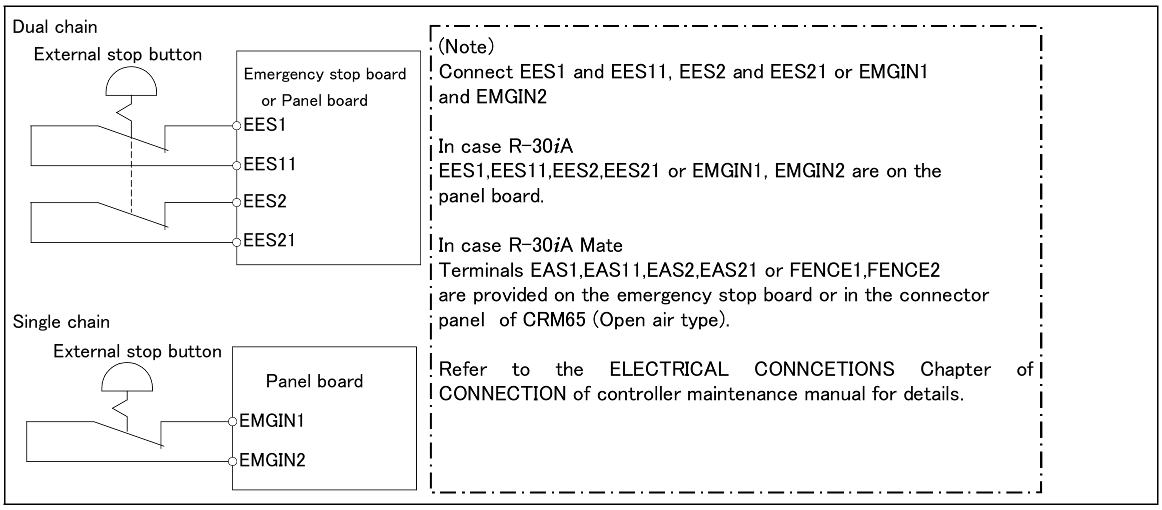
Fig.3 (c) Footstep for maintenance

3.1 OPERATOR SAFETY

The operator is a person who operates the robot system. In this sense, a worker who operates the teach pendant is also an operator. However, this section does not apply to teach pendant operators.

(1) If you do not have to operate the robot, turn off the power of the robot controller or press the EMERGENCY STOP button, and then proceed with necessary work.
(2) Operate the robot system at a location outside of the safety fence
(3) Install a safety fence with a safety gate to prevent any worker other than the operator from entering the work area unexpectedly and to prevent the worker from entering a dangerous area.
(4) Install an EMERGENCY STOP button within the operator's reach.

The robot controller is designed to be connected to an external EMERGENCY STOP button. With this connection, the controller stops the robot operation (Please refer to "STOP TYPE OF ROBOT" in SAFETY PRECAUTIONS for detail of stop type), when the external EMERGENCY STOP button is pressed. See the diagram below for connection.



(Note)
Connect EES1 and EES11, EES2 and EES21 or EMGIN1 and EMGIN2

In case R-30*i*A
EES1,EES11,EES2,EES21 or EMGIN1, EMGIN2 are on the panel board.

In case R-30*i*A Mate
Terminals EAS1,EAS11,EAS2,EAS21 or FENCE1,FENCE2 are provided on the emergency stop board or in the connector panel of CRM65 (Open air type).

Refer to the ELECTRICAL CONNCETIONS Chapter of CONNECTION of controller maintenance manual for details.

Fig.3.1 Connection diagram for external emergency stop button

3.2 SAFETY OF THE PROGRAMMER

While teaching the robot, the operator must enter the work area of the robot. The operator must ensure the safety of the teach pendant operator especially.

(1) Unless it is specifically necessary to enter the robot work area, carry out all tasks outside the area.
(2) Before teaching the robot, check that the robot and its peripheral devices are all in the normal operating condition.
(3) If it is inevitable to enter the robot work area to teach the robot, check the locations, settings, and other conditions of the safety devices (such as the EMERGENCY STOP button, the DEADMAN switch on the teach pendant) before entering the area.
(4) The programmer must be extremely careful not to let anyone else enter the robot work area.
(5) Programming should be done outside the area of the safety fence as far as possible. If programming needs to be done in the area of the safety fence, the programmer should take the following precautions:
  - Before entering the area of the safety fence, ensure that there is no risk of dangerous situations in the area.
  - Be prepared to press the emergency stop button whenever necessary.
  - Robot motions should be made at low speeds.
  - Before starting programming, check the entire system status to ensure that no remote instruction to the peripheral equipment or motion would be dangerous to the user.

Our operator panel is provided with an emergency stop button and a key switch (mode switch) for selecting the automatic operation mode (AUTO) and the teach modes (T1 and T2). Before entering the inside of the safety fence for the purpose of teaching, set the switch to a teach mode, remove the key from the mode switch to prevent other people from changing the operation mode carelessly, then open the safety gate. If the safety gate is opened with the automatic operation mode set, the robot stops (Please refer to "STOP TYPE OF ROBOT" in SAFETY PRECAUTIONS for detail of stop type). After the switch is set to a teach mode, the safety gate is disabled. The programmer should understand that the safety gate is disabled and is responsible for keeping other people from entering the inside of the safety fence. (In case of R-30*i*A Mate Controller standard specification, there is no mode switch. The automatic operation mode and the teach mode is selected by teach pendant enable switch.)

Our teach pendant is provided with a DEADMAN switch as well as an emergency stop button. These button and switch function as follows:
(1) Emergency stop button: Causes an emergency stop (Please refer to "STOP TYPE OF ROBOT" in SAFETY PRECAUTIONS for detail of stop type) when pressed.
(2) DEADMAN switch: Functions differently depending on the teach pendant enable/disable switch setting status.
  (a) Disable: The DEADMAN switch is disabled.
  (b) Enable: Servo power is turned off when the operator releases the DEADMAN switch or when the operator presses the switch strongly.
  Note) The DEADMAN switch is provided to stop the robot when the operator releases the teach pendant or presses the pendant strongly in case of emergency. The R-30*i*A/ R-30*i*A Mate employs a 3-position DEADMAN switch, which allows the robot to operate when the 3-position DEADMAN switch is pressed to its intermediate point. When the operator releases the DEADMAN switch or presses the switch strongly, the robot stops immediately.

The operator's intention of starting teaching is determined by the controller through the dual operation of setting the teach pendant enable/disable switch to the enable position and pressing the DEADMAN switch. The operator should make sure that the robot could operate in such conditions and be responsible in carrying out tasks safely.

Based on the risk assessment by FANUC, number of operation of DEADMAN SW should not exceed about 10000 times per year.

The teach pendant, operator panel, and peripheral device interface send each robot start signal. However the validity of each signal changes as follows depending on the mode switch and the DEADMAN switch of the operator panel, the teach pendant enable switch and the remote condition on the software.

In case of R-30*i*A controller or CE or RIA specification of R-30*i*A Mate controller

Mode	Teach pendant enable switch	Software remote condition	Teach pendant	Operator panel	Peripheral device
AUTO mode	On	Local	Not allowed	Not allowed	Not allowed
		Remote	Not allowed	Not allowed	Not allowed
	Off	Local	Not allowed	Allowed to start	Not allowed
		Remote	Not allowed	Not allowed	Allowed to start
T1, T2 mode	On	Local	Allowed to start	Not allowed	Not allowed
		Remote	Allowed to start	Not allowed	Not allowed
	Off	Local	Not allowed	Not allowed	Not allowed
		Remote	Not allowed	Not allowed	Not allowed

T1,T2 mode:DEADMAN switch is effective.

In case of standard specification of R-30*i*A Mate controller

Teach pendant enable switch	Software remote condition	Teach pendant	Peripheral device
On	Ignored	Allowed to start	Not allowed
Off	Local	Not allowed	Not allowed
	Remote	Not allowed	Allowed to start

(6) (Only when R-30*i*A Controller or CE or RIA specification of R-30*i*A Mate controller is selected.) To start the system using the operator's panel, make certain that nobody is the robot work area and that there are no abnormal conditions in the robot work area.
(7) When a program is completed, be sure to carry out a test operation according to the procedure below.
    (a) Run the program for at least one operation cycle in the single step mode at low speed.
    (b) Run the program for at least one operation cycle in the continuous operation mode at low speed.
    (c) Run the program for one operation cycle in the continuous operation mode at the intermediate speed and check that no abnormalities occur due to a delay in timing.
    (d) Run the program for one operation cycle in the continuous operation mode at the normal operating speed and check that the system operates automatically without trouble.
    (e) After checking the completeness of the program through the test operation above, execute it in the automatic operation mode.
(8) While operating the system in the automatic operation mode, the teach pendant operator should leave the robot work area.

3.3 SAFETY OF THE MAINTENANCE ENGINEER

For the safety of maintenance engineer personnel, pay utmost attention to the following.

(1) During operation, never enter the robot work area.
(2) A hazardous situation may arise when the robot or the system, are kept with their power-on during maintenance operations. Therefore, for any maintenance operation, the robot and the system should be put into the power-off state. If necessary, a lock should be in place in order to prevent any other person from turning on the robot and/or the system. In case maintenance needs to be executed in the power-on state, the emergency stop button must be pressed.
(3) If it becomes necessary to enter the robot operation range while the power is on, press the emergency stop button on the operator panel, or the teach pendant before entering the range. The

maintenance personnel must indicate that maintenance work is in progress and be careful not to allow other people to operate the robot carelessly.

(4) When entering the area enclosed by the safety fence, the maintenance worker must check the entire system in order to make sure no dangerous situations exist. In case the worker needs to enter the safety area whilst a dangerous situation exists, extreme care must be taken, and entire system status must be carefully monitored.

(5) Before the maintenance of the pneumatic system is started, the supply pressure should be shut off and the pressure in the piping should be reduced to zero.

(6) Before the start of teaching, check that the robot and its peripheral devices are all in the normal operating condition.

(7) Do not operate the robot in the automatic mode while anybody is in the robot work area.

(8) When you maintain the robot alongside a wall or instrument, or when multiple workers are working nearby, make certain that their escape path is not obstructed.

(9) When a tool is mounted on the robot, or when any moving device other than the robot is installed, such as belt conveyor, pay careful attention to its motion.

(10) If necessary, have a worker who is familiar with the robot system stand beside the operator panel and observe the work being performed. If any danger arises, the worker should be ready to press the EMERGENCY STOP button at any time.

(11) When replacing a part, please contact FANUC service center. If a wrong procedure is followed, an accident may occur, causing damage to the robot and injury to the worker.

(12) When replacing or reinstalling components, take care to prevent foreign material from entering the system.

(13) When handling each unit or printed circuit board in the controller during inspection, turn off the circuit breaker to protect against electric shock.
If there are two cabinets, turn off the both circuit breaker.

(14) A part should be replaced with a part recommended by FANUC. If other parts are used, malfunction or damage would occur. Especially, a fuse that is not recommended by FANUC should not be used. Such a fuse may cause a fire.

(15) When restarting the robot system after completing maintenance work, make sure in advance that there is no person in the work area and that the robot and the peripheral devices are not abnormal.

(16) When a motor or brake is removed, the robot arm should be supported with a crane or other equipment beforehand so that the arm would not fall during the removal.

(17) Whenever grease is spilled on the floor, it should be removed as quickly as possible to prevent dangerous falls.

(18) The following parts are heated. If a maintenance worker needs to touch such a part in the heated state, the worker should wear heat-resistant gloves or use other protective tools.

- Servo motor
- Inside the controller
- Reducer
- Gearbox
- Wrist unit

(19) Maintenance should be done under suitable light. Care must be taken that the light would not cause any danger.

(20) When a motor, reducer, or other heavy load is handled, a crane or other equipment should be used to protect maintenance workers from excessive load. Otherwise, the maintenance workers would be severely injured.

(21) The robot should not be stepped on or climbed up during maintenance. If it is attempted, the robot would be adversely affected. In addition, a misstep can cause injury to the worker.

(22)When performing maintenance work in high place, secure a footstep and wear safety belt.

(23) After the maintenance is completed, spilled oil or water and metal chips should be removed from the floor around the robot and within the safety fence.

(24) When a part is replaced, all bolts and other related components should put back into their original places. A careful check must be given to ensure that no components are missing or left not mounted.

(25) In case robot motion is required during maintenance, the following precautions should be taken :

- Foresee an escape route. And during the maintenance motion itself, monitor continuously the whole system so that your escape route will not become blocked by the robot, or by peripheral equipment.
- Always pay attention to potentially dangerous situations, and be prepared to press the emergency stop button whenever necessary.

(26) The robot should be periodically inspected. (Refer to the robot mechanical manual and controller maintenance manual.) A failure to do the periodical inspection can adversely affect the performance or service life of the robot and may cause an accident

(27) After a part is replaced, a test execution should be given for the robot according to a predetermined method. (See TESTING section of “Controller operator’s manual”.) During the test execution, the maintenance staff should work outside the safety fence.

4 SAFETY OF THE TOOLS AND PERIPHERAL DEVICES

4.1 PRECAUTIONS IN PROGRAMMING

(1) Use a limit switch or other sensor to detect a dangerous condition and, if necessary, design the program to stop the robot when the sensor signal is received.

(2) Design the program to stop the robot when an abnormal condition occurs in any other robots or peripheral devices, even though the robot itself is normal.

(3) For a system in which the robot and its peripheral devices are in synchronous motion, particular care must be taken in programming so that they do not interfere with each other.

(4) Provide a suitable interface between the robot and its peripheral devices so that the robot can detect the states of all devices in the system and can be stopped according to the states.

4.2 PRECAUTIONS FOR MECHANISM

(1) Keep the component cells of the robot system clean, and operate the robot in an environment free of grease, water, and dust.

(2) Don’t use unconfirmed liquid for cutting fluid and cleaning fluid.

(3) Employ a limit switch or mechanical stopper to limit the robot motion so that the robot or cable does not strike against its peripheral devices or tools.

(4) Observe the following precautions about the mechanical unit cables. When theses attentions are not kept, unexpected troubles might occur.

- Use mechanical unit cable that have required user interface.
- Don’t add user cable or hose to inside of mechanical unit.
- Please do not obstruct the movement of the mechanical unit cable when cables are added to outside of mechanical unit.
- In the case of the model that a cable is exposed, Please do not perform remodeling (Adding a protective cover and fix an outside cable more) obstructing the behavior of the outcrop of the cable.
- Please do not interfere with the other parts of mechanical unit when install equipments in the robot.

(5) The frequent power-off stop for the robot during operation causes the trouble of the robot. Please avoid the system construction that power-off stop would be operated routinely. (Refer to bad case example.) Please execute power-off stop after reducing the speed of the robot and stopping it by hold stop or cycle stop when it is not urgent. (Please refer to "STOP TYPE OF ROBOT" in SAFETY PRECAUTIONS for detail of stop type.)
(Bad case example)

- Whenever poor product is generated, a line stops by emergency stop.
- When alteration was necessary, safety switch is operated by opening safety fence and power-off stop is executed for the robot during operation.
- An operator pushes the emergency stop button frequently, and a line stops.
- An area sensor or a mat switch connected to safety signal operate routinely and power-off stop is executed for the robot.

(6) Robot stops urgently when collision detection alarm (SRVO-050) etc. occurs. The frequent urgent stop by alarm causes the trouble of the robot, too. So remove the causes of the alarm.

5 SAFETY OF THE ROBOT MECHANISM

5.1 PRECAUTIONS IN OPERATION

(1) When operating the robot in the jog mode, set it at an appropriate speed so that the operator can manage the robot in any eventuality.
(2) Before pressing the jog key, be sure you know in advance what motion the robot will perform in the jog mode.

5.2 PRECAUTIONS IN PROGRAMMING

(1) When the work areas of robots overlap, make certain that the motions of the robots do not interfere with each other.
(2) Be sure to specify the predetermined work origin in a motion program for the robot and program the motion so that it starts from the origin and terminates at the origin.
Make it possible for the operator to easily distinguish at a glance that the robot motion has terminated.

5.3 PRECAUTIONS FOR MECHANISMS

(1) Keep the work areas of the robot clean, and operate the robot in an environment free of grease, water, and dust.

5.4 PROCEDURE TO MOVE ARM WITHOUT DRIVE POWER IN EMERGENCY OR ABNORMAL SITUATIONS

For emergency or abnormal situations (e.g. persons trapped in or by the robot), brake release unit can be used to move the robot axes without drive power.
Please refer to controller maintenance manual and mechanical unit operator’s manual for using method of brake release unit and method of supporting robot.

6 SAFETY OF THE END EFFECTOR

6.1 PRECAUTIONS IN PROGRAMMING

(1) To control the pneumatic, hydraulic and electric actuators, carefully consider the necessary time delay after issuing each control command up to actual motion and ensure safe control.
(2) Provide the end effector with a limit switch, and control the robot system by monitoring the state of the end effector.

7 STOP TYPE OF ROBOT

The following three robot stop types exist:

Power-Off Stop (Category 0 following IEC 60204-1)

Servo power is turned off and the robot stops immediately. Servo power is turned off when the robot is moving, and the motion path of the deceleration is uncontrolled.
The following processing is performed at Power-Off stop.

- An alarm is generated and servo power is turned off.
- The robot operation is stopped immediately. Execution of the program is paused.

Controlled stop (Category 1 following IEC 60204-1)

The robot is decelerated until it stops, and servo power is turned off.
The following processing is performed at Controlled stop.

- The alarm "SRVO-199 Controlled stop" occurs along with a decelerated stop. Execution of the program is paused.
- An alarm is generated and servo power is turned off.

Hold (Category 2 following IEC 60204-1)

The robot is decelerated until it stops, and servo power remains on.
The following processing is performed at Hold.

- The robot operation is decelerated until it stops. Execution of the program is paused.

⚠ **WARNING**
The stopping distance and stopping time of Controlled stop are longer than the stopping distance and stopping time of Power-Off stop. A risk assessment for the whole robot system, which takes into consideration the increased stopping distance and stopping time, is necessary when Controlled stop is used.

When the emergency stop button is pressed or the FENCE is open, the stop type of robot is Power-Off stop or Controlled stop. The configuration of stop type for each situation is called *stop pattern*. The stop pattern is different according to the controller type or option configuration.

There are the following 3 Stop patterns.

Stop pattern	Mode	Emergency stop button	External Emergency stop	FENCE open	SVOFF input	Servo disconnect
A	AUTO	P-Stop	P-Stop	C-Stop	C-Stop	P-Stop
	T1	P-Stop	P-Stop	-	C-Stop	P-Stop
	T2	P-Stop	P-Stop	-	C-Stop	P-Stop
B	AUTO	P-Stop	P-Stop	P-Stop	P-Stop	P-Stop
	T1	P-Stop	P-Stop	-	P-Stop	P-Stop
	T2	P-Stop	P-Stop	-	P-Stop	P-Stop
C	AUTO	C-Stop	C-Stop	C-Stop	C-Stop	C-Stop
	T1	P-Stop	P-Stop	-	C-Stop	P-Stop
	T2	P-Stop	P-Stop	-	C-Stop	P-Stop

P-Stop:	Power-Off stop
C-Stop:	Controlled stop
-:	Disable

The following table indicates the Stop pattern according to the controller type or option configuration.

Option	R-30*i*A				R-30*i*A Mate		
	Standard (Single)	Standard (Dual)	RIA type	CE type	Standard	RIA type	CE type
Standard	B (*)	A	A	A	A (**)	A	A
Stop type set (Stop pattern C) (A05B-2500-J570)	N/A	N/A	C	C	N/A	C	C

(*) R-30*i*A standard (single) does not have servo disconnect.

(**) R-30*i*A Mate Standard does not have servo disconnect, and the stop type of SVOFF input is Power-Off stop.

The stop pattern of the controller is displayed in "Stop pattern" line in software version screen. Please refer to "Software version" in operator's manual of controller for the detail of software version screen.

"Controlled stop by E-Stop" option

When Stop type set (Stop pattern C) (A05B-2500-J570)) is specified, the stop type of the following alarms becomes Controlled stop but only in AUTO mode. In T1 or T2 mode, the stop type is Power-Off stop which is the normal operation of the system.

Alarm	Condition
SRVO-001 Operator panel E-stop	Operator panel emergency stop is pressed.
SRVO-002 Teach pendant E-stop	Teach pendant emergency stop is pressed.
SRVO-007 External emergency stops	External emergency stop input (EES1-EES11, EES2-EES21) is open. (R-30*i*A controller)
SRVO-194 Servo disconnect	Servo disconnect input (SD4-SD41, SD5-SD51) is open. (R-30*i*A controller)
SRVO-218 Ext.E-stop/Servo Disconnect	External emergency stop input (EES1-EES11, EES2-EES21) is open. (R-30*i*A Mate controller)
SRVO-408 DCS SSO Ext Emergency Stop	In DCS Safe I/O connect function, SSO[3] is OFF.
SRVO-409 DCS SSO Servo Disconnect	In DCS Safe I/O connect function, SSO[4] is OFF.

Controlled stop is different from Power-Off stop as follows:

- In Controlled stop, the robot is stopped on the program path. This function is effective for a system where the robot can interfere with other devices if it deviates from the program path.

- In Controlled stop, physical impact is less than Power-Off stop. This function is effective for systems where the physical impact to the mechanical unit or EOAT (End Of Arm Tool) should be minimized.
- The stopping distance and stopping time of Controlled stop is longer than the stopping distance and stopping time of Power-Off stop, depending on the robot model and axis. Please refer to the operator's manual of a particular robot model for the data of stopping distance and stopping time.

In case of R-30*i*A or R-30*i*A Mate, this function is available only in CE or RIA type hardware.

When this option is loaded, this function cannot be disabled.

The stop type of DCS Position and Speed Check functions is not affected by the loading of this option.

⚠ **WARNING**

The stopping distance and stopping time of Controlled stop are longer than the stopping distance and stopping time of Power-Off stop. A risk assessment for the whole robot system, which takes into consideration the increased stopping distance and stopping time, is necessary when this option is loaded.

TABLE OF CONTENTS

1 KAREL LANGUAGE OVERVIEW

1.1 OVERVIEW

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 and concurrent motion support

The SYSTEM R-30*i*A 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.

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

1.2 KAREL PROGRAMMING LANGUAGE

1.2.1 Overview

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 Word Pad, however, text editor can not translate the KAREL program before executing it.

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 by ROBOGUIDE or OLPC PRO. After translated, the resulting p-code program can be loaded onto the controller using the KAREL Command Language (KCL), or the FILE menu.

> ⚠ **CAUTION**
> Basically, translate a KAREL program by ROBOGUIDE. Refer to the APPENDIX F "Translating by 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 a single taught position 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 motion statements, I/O statements, and routine calls.
The program development cycle is described briefly in the following list. Subsection 1.2.2 - Subsection 1.2.6 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 execute 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 of programs are sample in this manual. Before installing and executing them, confirm them on your responsibility.
2. Please note that in some cases executing KAREL program affects the performance of the controller.
3. Be careful not to delete needed programs, variables, files, data and so on.
4. ROBOGUIDE is useful for translate and debug a KAREL program. However, confirm it on the real robot.

1.2.2 Creating a Program

Create a KAREL program by ROBOGUIDE basically. It can be also created off-line editor such as OLPC PRO, text editor such as WordPad. The resulting file is called the source file or source code.

1.2.3 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.
Basically, translate a KAREL program by ROBOGUIDE. You can invoke the translator from ROBOGUIDE (also OLPC PRO), 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 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.

⚠ **CAUTION**
Basically, translate a KAREL program by ROBOGUIDE. Refer to the chapter "Translating by ROBOGUIDE".

1.2.4 Loading Program Logic and Data

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.

1.2.5 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 the robot moves the way it should.
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 subsection 5.1.6 .

You can run KAREL program directly, as a task. Or you can run TP program and call the KAREL program. Generally speaking, execution by the former way (direct RUN as a task) is faster than the latter (call from TP program).

1.2.6 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.

1.2.7 Program Structure

A KAREL program is composed of declaration and executable sections made up of KAREL language statements, as shown in Example 1.2.7 .

Example 1.2.7 Structure of a KAREL Program

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

In Example 1.2.7 , 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 subsection 2.1.3 .

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 Example 1.2.7). Identifiers are described in subsection 2.1.4 .

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 is 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 Example 1.2.7), 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 , Chapter 3 , and Chapter 5 .

1.3 SYSTEM SOFTWARE

The R-30*i*A system includes 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 Mechanical Unit Manual specific for your robot and controller model.

1.3.1 Software Components

R-30*i*A system software is the FANUC supplied software that is executed by the controller CPU, which allows you to operate the R-30*i*A 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:

- **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
- **Teach Pendant Screens -** screens that facilitate operation of the KAREL system
- **KCL -** KAREL Command Language
- **KAREL Interpreter -** executes KAREL programs

See Also: application-specific Operator's Manual for detailed operation procedures using teach pendant screens.

1.3.2 Supported Robots

The robot, using the appropriate tooling, performs application tasks directed by the system software and controller. The R-30*i*A system supports a variety of robots, each designed for a specific type of application.
For a current list of supported robot models, consult your FANUC service center.

1.4 CONTROLLER

The R-30*i*A controller contains the electronic circuitry and memory required to operate the R-30*i*A system. The electronic circuitry, supported by the system software, directs the operation 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.

1.4.1 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 the 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 start up 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.

NOTE

If you have a 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 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.

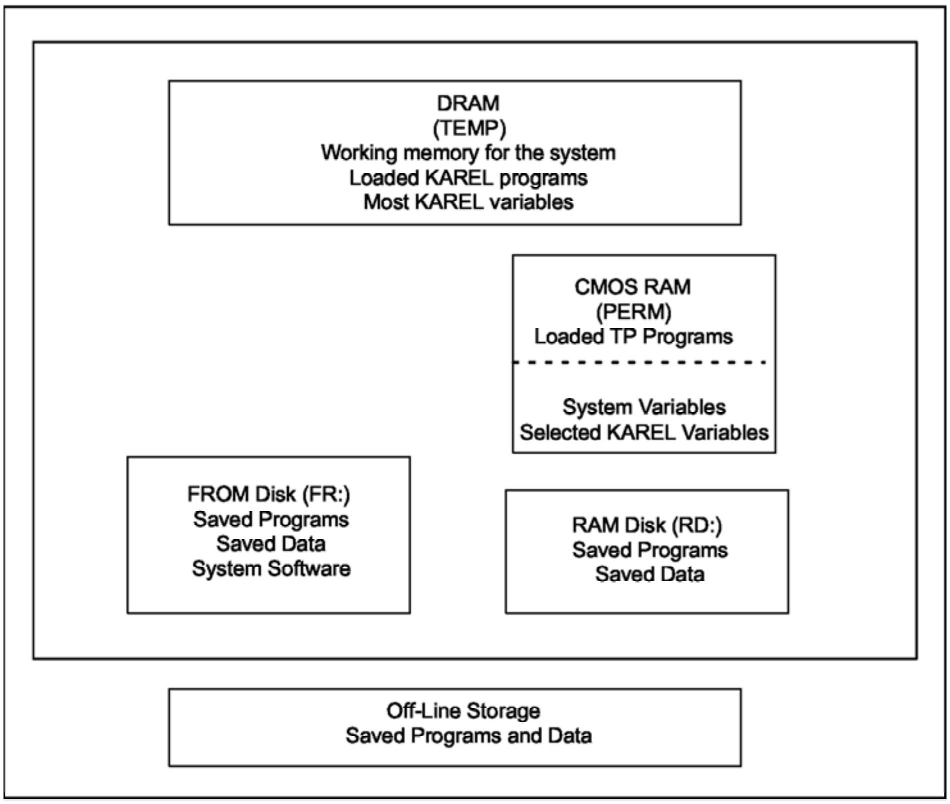


Fig. 1.4.1 Controller memory

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

1.4.2 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 11 , for more information

1.4.3 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 1.4.3 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 11 "INPUT/OUTPUT SYSTEM".

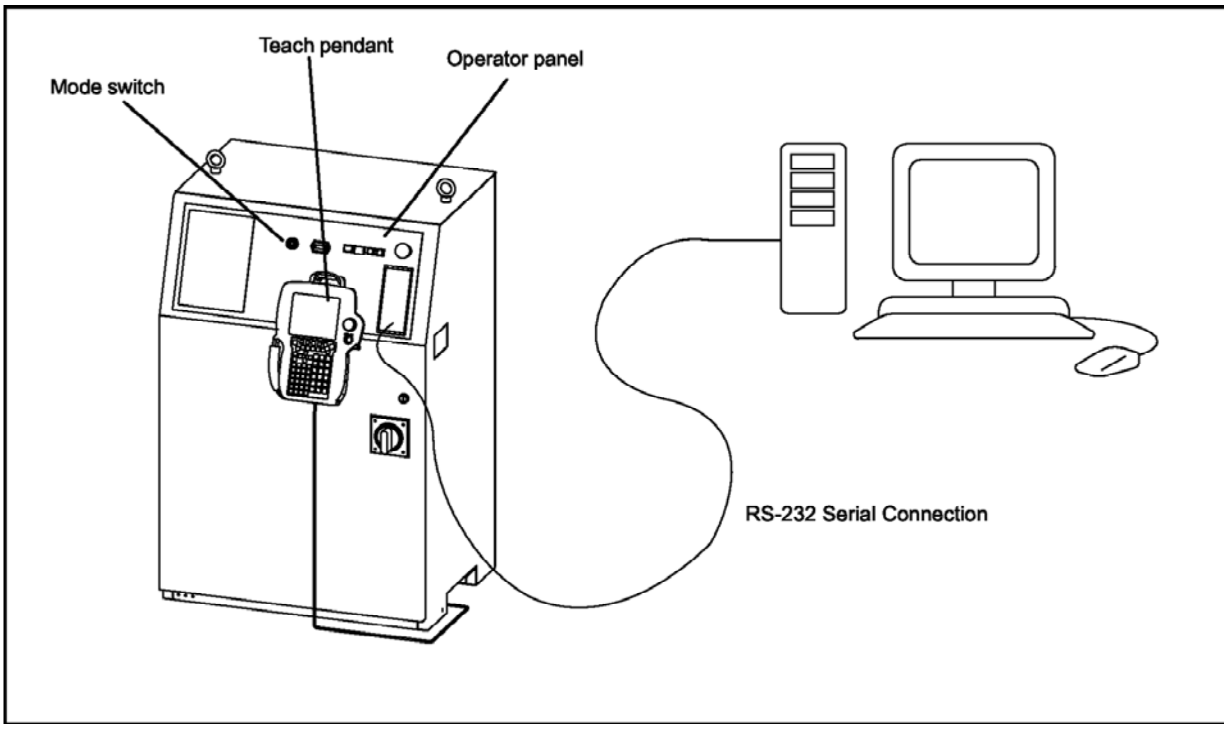


Fig. 1.4.3 R-30*i*A controller

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.

The application-specific Operator's Manual provides descriptions of each of the user interface devices, as well as procedures for operating each device.

1.4.4 Frame

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 program. These values must be set some adequate value by KAREL program before invoking the function that treats the position data.

Current selected USER or TOOL frame is used on the motion instruction of TP program. In the case that 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 the KAREL program does not have the USER or TOOL frame number at the time of teaching differed from 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 the value of WORLD or Mechanical interface coordinate.

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 lists the elements in the ASCII character set. Three character sets are available in the KAREL language:
See Also: CHR Built-In Procedure, Appendix A .

Table 2.1.1 ASCII character set

TYPE	CHARACTER
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 &.

See Also: Appendix D 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

TYPE	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 , 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	DOWNTO	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: Index for references to descriptions of KAREL reserved words

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
- Can have only one meaning within a particular scope. Refer to Subsection 5.1.4 .
- Cannot be reserved words
- Must be defined before they can be used.

For example, the program excerpt in Example 2.1.4 shows how to declare program, variable, and constant identifiers.

Example 2.1.4 Declaring Identifiers

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

2.1.5 Labels

Labels are special identifiers that mark places in the program to which program control can be transferred using the GOTO 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 Example 2.1.5, **weld: :** denotes the section of the program in which a part is welded. When the statement **go to weld** is executed, program control is transferred to the **weld** section.

Example 2.1.5 Using labels

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

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

Table 2.1.6(b) Port and file predefined identifier summary

Predefined Identifier	Type
DIN (Digital input) DOUT (Digital output)	Boolean port array
GIN (Group input) GOUT (Group output) AIN (Analog input) AOUT (Analog output)	Integer port array
TPIN (Teach pendant input) 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)	Boolean port array
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, Example 2.1.8 contains some comments from Example 2.2(a) and Example 2.2(b).

Example 2.1.8 Comments from within a program

```
--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
```

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 A 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 SELECT 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.

Directive	Description
%NOPAUSESHFT	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.
%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.

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

Example 2.2 (a) %INCLUDE directive in a KAREL program

```
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
    $SPEED = 200.0
    $MOTYPE = LINEAR
    OPEN HAND gripper
-- Loop to move total number of objects
    FOR count = 1 TO num_of_parts DO
        MOVE TO original
        CLOSE HAND gripper
        MOVE TO destination
        OPEN HAND gripper
    ENDFOR                      -- End of loop
END mover
```

Example 2.2 (b) Include file mover_decs for a KAREL program

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

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 Subsection 2.4 .

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 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.

Example 2.4.1 shows an example of user-defined data type usage and continuation character usage.

Example 2.4.1 User-Defined data type example

```
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
```

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 Example 2.4.2(a).

Example 2.4.2 (a) Defining a data type as a user-defined structure

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

- 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 Example 2.4.2(b) .

Example 2.4.2 (b) Accessing elements of a user-defined structure in a karel program

```
var_name = new_type_name.field_nam_1
new_type_name.field_name_2 = expression
outer_struct_name.inner_struct_name&
  .field_name = expression
```

- 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 Example 2.4.2(c).

Example 2.4.2 (c) Defining a range of executable statements

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

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 Example 2.4.2(d).

Example 2.4.2 (d) Valid STRUCTURE definitions

```
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
```

- 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 Example 2.4.2(e).

Example 2.4.2 (e) Defining a variable as a type previously defined as a structure

```
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
```

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 KAREL variable screen.
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 Example 2.5.1(a) .

Example 2.5.1 (a) Declaring variables as arrays with two or three dimensions

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

- Access elements of these arrays in KAREL statements. See Example 2.5.1(b) .

Example 2.5.1 (b) Accessing elements of multi-dimensional arrays in KAREL statements

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

- Declare routine parameters as multi-dimensional arrays. See Example 2.5.1(c) .

Example 2.5.1 (c) Declaring routine parameters as multi-dimensional arrays

```
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)
```

- Access elements with KCL commands and the teach pendant.
- Show elements in 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 Example 2.5.1(d).

Example 2.5.1 (d) Using a subarray

```
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
```

- 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 Example 2.5.2.

Example 2.5.2 Indicates that the size of an array is "to-be-determined"

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

- 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.

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 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**	**#**	**@**	**:**
Types of Operators											
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	–	–	–	–	–

Operator	+	-	*	/	DIV MOD	< >,>= <=, <, >, =	> =<	AND OR NOT	#	@	:
Types of Operators											
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 such as DOUT[1]. The expression can be any valid KAREL expression. The following examples are acceptable assignments:

```
$KAREL_ENB = 1 -- assigns a INTEGER value to a system variable
real_var = real_var + 1 -- assigns an REAL value to an 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: Relational Operations, ROUND and TRUNC built-in functions, Appendix A, ``KAREL Language Alphabetical Description"

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 5 OR 8	0101 AND 1000 0101 OR 1000	0000 = 0 1101 = 13
-4 AND 8 -4 OR 8	1100 AND 1000 1100 OR 1000	1000 = 8 1100 = -4
NOT 5 NOT -15	NOT 0101 NOT 110001	1010 = -6* 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
:, @, #	↓
*, /, AND, DIV, MOD	↓
Unary + and -, OR, +, -	↓
<, >, =, < >, < =, > =, > = <	Low

3.2.2 Relational Operations

Relational operators (< >, =, >, <, <=, >=) 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 (<>) 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 (<>) 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 >= or <= 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 = 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
	FALSE	TRUE	
		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 Subsection 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.
- The relational operator (>=<) 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 (>=<) is used to determine if the current robot position (determined by using the CURPOS built-in procedure) is near the designated perch position:

Example 3.2.4 (a) Relational operator

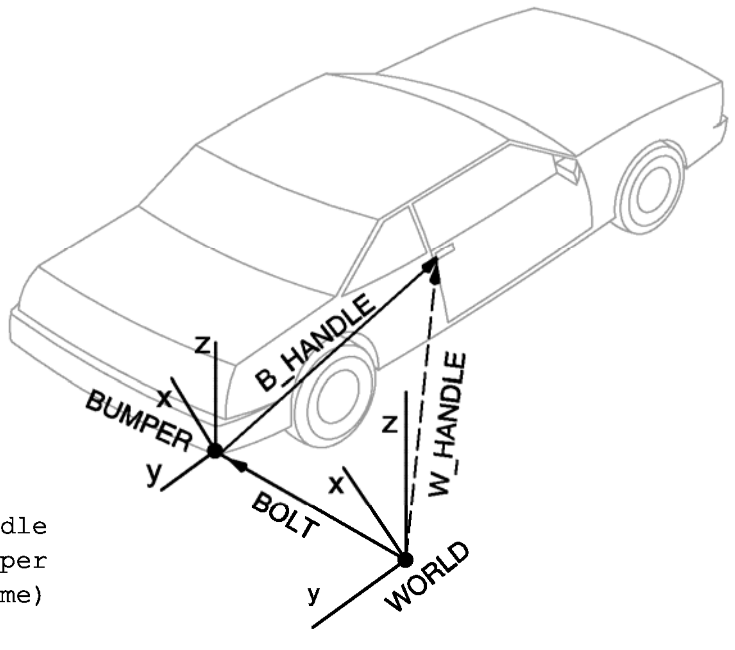
```
IF perch >=< CURPOS (0,0) THEN
  WRITE('OK', CR)
ELSE
      ABORT
ENDIF
```

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.



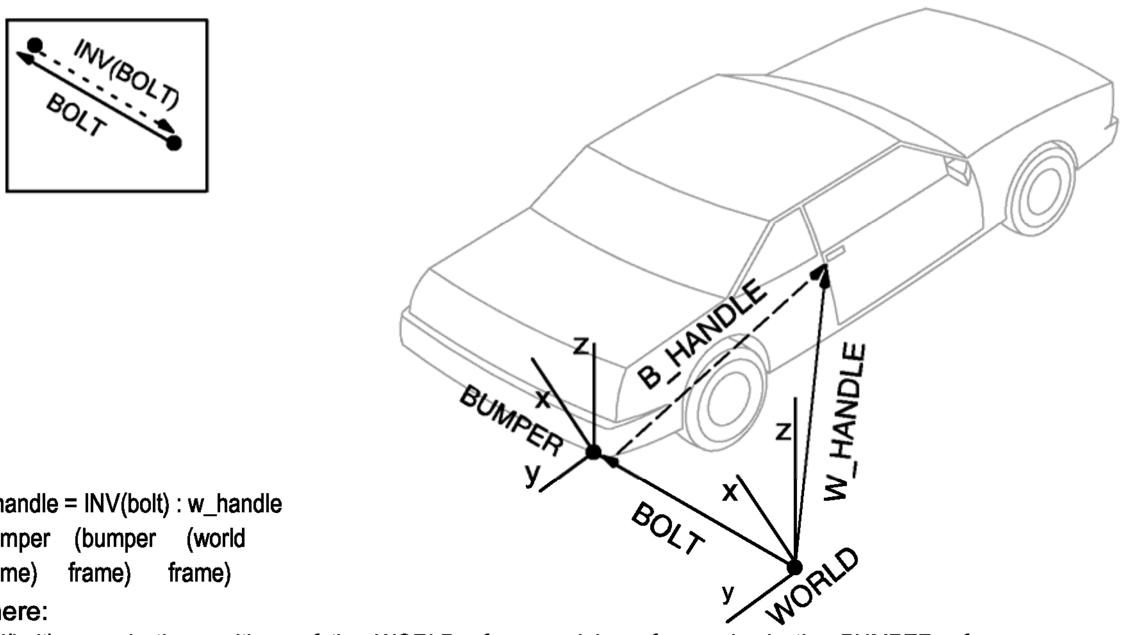
```
w_handle = bolt : b_handle
(world    (world   (bumper
 frame)    frame)   frame)
```

where:

`bolt` **is the position of the BUMPER frame origin referenced in the WORLD frame.**
`w_handle` **is the handle position referenced in the WORLD frame.**
`b_handle` **is the handle position referenced in the BUMPER frame.**

Fig. 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) .



b_handle = INV(bolt) : w_handle
(bumper frame) (bumper frame) (world frame)

where:

INV(bolt)	is the position of the WORLD frame origin referenced in the BUMPER frame.
w_handle	is the handle position referenced in the WORLD frame.
b_handle	is the handle position referenced in the BUMPER frame.

Fig. 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: INV Built-In Function, Appendix A .

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:

Example3.4.2 (b) Vector operations

```
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
```

- 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 / 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 SIN(Θ), whereΘis the angle between the two vectors. For example, VEC(3.0,4.0,5.0) # 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, VEC(3.0,4.0,5.0) @ 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
VEC(3.0,7.0,6.0) + VEC(12.6,3.2,7.5)	(15.6,10.2,13.5)
VEC(7.6,9.0,7.0) - VEC(14.0,3.5,17.0)	(-6.4,5.5,-10)
4.5 * VEC(3.2,7.6,4.0)	(14.4,34.2,18.0)
VEC(12.7,2.0,8.3) * 7.6	(96.52,15.2,63.08)
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 Example 3.2.4(c).

Example 3.2.4 (c) Specifying a KAREL routine to return a STRING value

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

- An operator can be used between strings to indicate the concatenation of the strings. See Example 3.2.4(d).

Example 3.2.4 (d) Using an operator to concatenate strings

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

- STRING expressions can be used in WRITE statements. See Example 3.2.4(e).

Example 3.2.4 (e) Using a STRING expression in a WRITE statement

```
WRITE(CHR(13) + string_1 + string_2)
```

- 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 Example 3.2.4(f).

Example 3.2.4 (f) String comparison

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

- You can build a string from another string. See Example 3.2.4(g).

Example 3.2.4 (g) Building a string from another string

```
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
```

4 PROGRAM CONTROL

4.1 OVERVIEW

Program control structures determine the flow of program or routine. Control structures include alternation control, looping control and unconditional branch statements.

4.2 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."

4.2.1 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: IF...THEN Statement, Appendix A , SELECT Statement, Appendix A .

4.2.2 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: FOR Statement, Appendix A , REPEAT Statement, Appendix A , WHILE Statement, Appendix A

4.2.3 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**
DO NOT 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: GO TO Statement, Appendix A .

4.2.4 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: ABORT Statement, DELAY Statement, PAUSE Statement, WAIT FOR Statement, all in Appendix A , Chapter 6 "CONDITION HANDLERS"

4.2.5 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 the chapter 6 to see the detail.

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 Subsection 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:
  - 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 Example 5.1.1(a).

Example 5.1.1 (a) Defining local routines using a FROM clause

```
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
```

- 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 Example 5.1.1(b) illustrate local and external procedure routine declarations.

Example 5.1.1 (b) Local and external procedure declarations

```
PROGRAM procs_lib
ROUTINE wait_a_bit
 --local procedure, no parameters
    BEGIN
    DELAY 20
 END wait_a_bit
ROUTINE move_there(p: POSITION)
 --local procedure, one parameter
    BEGIN
    MOVE TO p        --reference to parameter p
 END move_there
ROUTINE calc_dist(p1,p2: POSITION; dist: REAL)
 FROM math_lib
 --external procedure defined in math_lib.kL
```

The example in Example 5.1.1(c) illustrates local and external function routine declarations.

Example 5.1.1 (c) Function 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
```

See Also: FROM Clause, ROUTINE Statement, Appendix A .

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.

Example 5.1.2(a) shows the declarations for two procedures followed by the procedure calls to invoke them.

Example 5.1.2 (a) Procedure calls

```
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
```

- 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.

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

Example 5.1.2 (b) Function calls

```
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
```

- 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 Example 5.1.2(c) , calls itself to calculate a factorial value.

Example 5.1.2 (c) Recursive function

```
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
```

- 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: Subsection 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.

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

Example 5.1.3 (a) Procedure RETURN statements

```
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
```

- 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
- 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 Example 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.

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

Example 5.1.3 (b) Function RETURN statements

```
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 compareb
```

See Also: ROUTINE Statement, Appendix A .

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.

Example 5.15(a) shows an example of a routine declaration and three calls to that routine.

Example 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.
- 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.

Example 5.1.5(b) shows a routine that affects the argument being passed to it differently depending on how the variable argument is passed.

Example 5.1.5 (b) Passing variable arguments

```
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
```

The output from the program in Example 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 Example 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))
    Refer to Appendix A for more information about BYNAME.
- 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.

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

Example 5.1.5 (c) Correct passage of an ARRAY

```
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
```

Example 5.15(d) shows the incorrect use of an ARRAY passed between two function routines.

Example 5.1.5 (d) Incorrect passage of an ARRAY

```
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
    print_ary(rtn_ary)
  END wrong
```

See Also: ARRAY_LEN Built-In Function, STR_LEN Built-In Function, Appendix A , 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 ARRAY OF BOOLEAN	1	2 not allowed	1 1 + array size
ARRAY OF BYTE	1	not allowed	1 + (array size)/4
CAM_SETUP ARRAY OF CAM_SETUP	1	not allowed not allowed	not allowed not allowed
CONFIG ARRAY OF CONFIG	1	2 not allowed	1 1 + array size
INTEGER ARRAY OF INTEGER	1	2 not allowed	1 1 + array size
FILE ARRAY OF FILE	1	not allowed not allowed	not allowed not allowed
JOINTPOS ARRAY OF JOINTPOS	2 1	12 not allowed	10 1 + 10 * array size
JOINTPOS1 ARRAY OF JOINTPOS1	2 1	4 not allowed	2 1 + 2 * array size
JOINTPOS2 ARRAY OF JOINTPOS2	2 1	5 not allowed	3 1 + 3 * array size
JOINTPOS3 ARRAY OF JOINTPOS3	2 1	6 not allowed	4 1 + 4 * array size
JOINTPOS4 ARRAY OF JOINTPOS4	2 1	7 not allowed	5 1 + 5 * array size
JOINTPOS5 ARRAY OF JOINTPOS5	2 1	8 not allowed	6 1 + 6 * array size
JOINTPOS6 ARRAY OF JOINTPOS6	2 1	9 not allowed	7 1 + 7 * array size
JOINTPOS7 ARRAY OF JOINTPOS7	2 1	10 not allowed	8 1 + 8 * array size
JOINTPOS8 ARRAY OF JOINTPOS8	2 1	11 not allowed	9 1 + 9 * array size
JOINTPOS9 ARRAY OF JOINTPOS9	2 1	12 not allowed	10 1 + 10 * array size
MODEL ARRAY OF MODEL	1 1	not allowed not allowed	not allowed not allowed
PATH	2	not allowed	not allowed
POSITION ARRAY OF POSITION	2 1	16 not allowed	14 1 + 14 * array size
REAL ARRAY OF REAL	1 1	2 not allowed	1 1 + array size
ARRAY OF SHORT	1	not allowed	1 + (array size)/2
STRING ARRAY OF STRING	2 1	2 + (string length+2)/4not allowed	(string length+2)/4 1+((string length+2) *array size)/4

Type	Parameter Passed by Reference	Parameter Passed by Value	Local Variable
VECTOR ARRAY OF VECTOR	1 1	4 not allowed	3 1 + 3 * array size
VIS_PROCESS ARRAY OF VIS_PROCESS	1 1	not allowed not allowed	not allowed not allowed
XYZWPR ARRAY OF XYZWPR	2 1	10 not allowed	8 1 + 8 * array size
XYZWPREXT ARRAY OF XYZWPREX	2 1	13 not allowed	11 1 + 11 * array size
ARRAY [m,n] OF some_type	1	not allowed	m(ele size/4 * n + 1)+1
ARRAY [l,m,n] OF some_type	1	not allowed	l(m(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 application-specific Operator's Manual. 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.
See Also: Appendix A , which lists optional KAREL built-ins and where they are documented.

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. Condition handlers are used to monitor and act on conditions throughout an entire program.
These 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

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	ENABLE CONDITION[n]
port_id[n] = expression	DISABLE CONDITION[n]
routine_name	PULSE DOUT[n] FOR t
SIGNAL EVENT[n]	UNPAUSE
NOABORT	ABORT
NOMESSAGE	CONTINUE
NOPAUSE	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 conds 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
n <= $COND_TIME	$COND_TIME
$COND_TIME < n <= (2 * $COND_TIME)	(2 * $COND_TIME)
(2 * $COND_TIME) < n <= (4 * $COND_TIME)	(4 * $COND_TIME)
(4 * $COND_TIME) < n <= (8 * $COND_TIME)	(8 * $COND_TIME)
(8 * $COND_TIME) < n <= (16 * $COND_TIME)	(16 * $COND_TIME)
(16 * $COND_TIME) < n <= (32 * $COND_TIME)	(32 * $COND_TIME)
(32 * $COND_TIME) < n <= (64 * $COND_TIME)	(64 * $COND_TIME)
(64 * $COND_TIME) < n <= (128 * $COND_TIME)	(128 * $COND_TIME)
(128 * $COND_TIME) < n <= (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.

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

Example 6.1.1 (a) Global condition handler definitions

```
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
```

- 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 Example 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.

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

Example 6.1.1 (b) Using 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
```

6.2 CONDITIONS

One or more conditions are specified in the condition list of a WHEN 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 clause are simultaneously satisfied, the condition handler is triggered and corresponding actions are performed.
    - If all of the conditions of an OR, WHEN 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 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: EVAL Clause, Appendix A .

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 etc). 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 port_id[n], or a variable. Operands on the right can be a variable, a constant, or an EVAL clause. (state)
operand < > operand	
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.

See Also: EVAL Clause, Appendix A .

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, 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)

CONDITION	SATISFIED (TRUE) WHEN
ABORT	The program is aborted. (event)
PAUSE	The program is paused. (event)
CONTINUE	The program is continued. (event)
SEMAPHORE[n]	The value of the semaphore specified by n is posted.

The following rules 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 errorccc 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.
- 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.

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: In Appendix A :
ABORT Condition
CONTINUE Condition
ERROR Condition
EVENT Condition
PAUSE Condition
SEMAPHORE Condition
application-specific Operator's Manual for error codes.

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: Actions and Statements, Appendix A .

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 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: WITH Clause, Appendix A, ``KAREL Language Alphabetical Description," for more information on $PRIORITY

6.3.3 Miscellaneous Actions

Table 6.3.3 describes other allowable actions.

Table 6.3.3 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.

ACTION	RESULT
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.
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 for more information on each miscellaneous action.

7 FILE INPUT/OUTPUT OPERATIONS

7.1 OVERVIEW

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.

7.2 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 7.2 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 7.2 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. Example 7.2 shows an example of declaring a FILE variable and of using FILE in the executable section of a program.

Example 7.2 Using FILE in a KAREL 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
```

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.

7.3 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.

7.3.1 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 7.3.1(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 for more information.

Table 7.3.1(a) Predefined attribute types

ATTRIBUTE TYPE	FUNCTION	SET_PORT_ATR OR SET_FILE_ATR	TP/ CRT	SERIAL PORTS	DATA FILES	PIPES	SOCKET MESSAGING
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
ATR_FIELD	Field	SET_FILE_ATR	valid	valid	valid	valid	valid
ATR_IA	Interactively write	SET_FILE_ATR	valid	valid	valid	valid	valid
ATR_PARITY	Parity	SET_PORT_ATR	not used	valid	not used	not used	not used

ATTRIBUTE TYPE	FUNCTION	SET_PORT_ATR OR SET_FILE_ATR	TP/ CRT	SERIAL PORTS	DATA FILES	PIPES	SOCKET MESSAGING
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_PIPWAIT	Wait for data	SET_FILE_ATR	not used	not used	not used	valid	valid

Table 7.3.1(b) contains detailed explanations of each attribute.

Table 7.3.1(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
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 Appendix D , for a listing of valid attribute values.	PORT	Read/ Write	Any ASCII character code	13 (carriage return)

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
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
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_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)
ATR_REVERSE Reverse transfer	The bytes will be swapped.	PORT, FILE	Read/ Write	Ignored	Not reverse transfer

Attribute Type	Description	Valid Device	Usage Mode	Valid Values	Default Value
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

7.3.2 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 for 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 8 , for a description of file names and file types.

7.3.3 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 7.3.3 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 7.3.3 Usage specifiers

SPECIFIER	FUNCTION	TP/CRT	PORTS	FILES	PIPES
RO	● Permits only read operations ● Sets file position to beginning of file ● File must already exist	valid	valid	valid	valid
RW	● Overwrites over existing data in a file, deleting existing data ● Permits read and write operations ● Sets file position to beginning of file ● File will be created if it does not exist	valid	valid	valid No use on FRx:	valid
AP	● Appends to end of existing data ● Permits read and write (First operation must be a write.) ● Sets file position to end of file ● File will be created if it does not exist	no use	valid	valid -RAM disk* no use on FRx:	valid
UD	● Updates from beginning of existing data. (Number of characters to be written must equal number of characters to be replaced.) ● Overwrites the existing data with the new data ● Permits read and write ● Sets file position to beginning of existing file	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 8, for more information on the RAM disk and Pipe devices.

Example 7.3.3 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.

Example 7.3.3 File string examples

```
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
```

See Also: Chapter 8 , for more information on the available storage devices

7.4 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.

Example 7.4 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.

Example 7.4 CLOSE FILE example

```
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
```

See Also: CLOSE FILE Statement, IO_STATUS Built-In Function, Appendix A for a description of errors.

7.5 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.

```
READ < file_var > (data_item {,data_item})

file_var : a FILE variable
data_item : a variable identifier and its optional format specifiers or the
reserved word CR
```

Reading data, data_item is listed as a part of 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 7.3.3).
- 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.

Example 7.5 shows several examples of the READ statement using a variety of file variables and data lists.

Example 7.5 READ statement examples

```
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
```

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:

- READ Statement, Appendix A .
- IO_STATUS Built-In Functions, Appendix A for a list of I/O error messages
- %CRTDEVICE Translator Directive, Appendix A .

7.6 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.

```
WRITE <file_var> (data_item { ,data_item })


file_var : a FILE variable
data_item : an expression and its optional format specifiers or the reserved
word CR
```

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 7.3.3).

- 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.

Example 7.6 shows several examples of the WRITE statement using a variety of file variables and data lists.

Example 7.6 WRITE statement examples

```
WRITE TPPROMPT('Press T.P. key "GO" when ready')
  WRITE TPFUNC ('  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
```

See Also: WRITE Statement, Appendix A . IO_STATUS Built-In Functions, Appendix A .

7.7 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 127 bytes.

7.8 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.
- 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 7.8(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 Subsection 7.8.1 , through Subsection 7.8.6 .

Table 7.8(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 7.8(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 Subsection 7.8.1 through Subsection 7.8.6 .

Table 7.8(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
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

7.8.1 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 7.8.1(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 7.8.1(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 for a file or 1 to 126 for other output devices.
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 7.8.1(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 7.8.1(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)
-1::9::16	" **FFFFFFFF**"	Hexadecimal output

7.8.2 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 7.8.2(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 7.8.2(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)(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 for a file or 1 to 126 for other output devices.
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). If it is negative, scientific notation is used.

Table 7.8.2(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 7.8.2(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

7.8.3 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 7.8.3(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 7.8.3(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 for a file or 1 to 126 for other output devices.

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 7.8.3(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 7.8.3(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

7.8.4 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 7.8.4(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 7.8.4(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)

DATA ITEM	INPUT DATA	RESULT
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.
The STRING must not be over 127 bytes in length for files or 126 bytes in length for other output devices. Otherwise, the program will be aborted with the ``STRING TOO LONG" error.
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 must be in the range of 1 to 127 for a file or 1 to 126 for other output devices.
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 7.8.4(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 7.8.4(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.

7.8.5 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 7.8.5 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: Subsection 7.8.2 , ``Formatting REAL Data Items," for information on the default output format and format specifiers used with REAL data items

Table 7.8.5 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**"

7.8.6 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 Subsection 7.8.2 .
The configuration string is not terminated with a CR, meaning you can follow it with other data on the same line.
Table 7.8.6 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 7.8.6 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: Subsection 7.8.2 , for information on format specifiers used with REAL data items
POS Built-In Function, Appendix A .

7.9 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
```

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, and VECTOR 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 7.9 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. Subsection 7.9.1 through Subsection 7.9.9 explain the effects of format specifiers on each data type in more detail.
See Also: SET_FILE_ATR Built-In Routine, Appendix A .

Table 7.9 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	56 bytes read or written	Ignored	Ignored
XYZWPR	32 bytes read or written	Ignored	Ignored
XYZWPREXT	44 bytes read or written	Ignored	Ignored
JOINTPOSn	4 + n*4 bytes read or written	Ignored	Ignored

7.9.1 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.

7.9.2 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.

7.9.3 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.

7.9.4 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.

7.9.5 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.

7.9.6 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 56 bytes long.

7.9.7 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 32 bytes long.

7.9.8 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 44 bytes long.

7.9.9 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 4 bytes plus 4 bytes for each axis.

7.10 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.

7.10.1 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 7.10.1 are defined for "t_sc".

Table 7.10.1 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 7.10.1(a) .

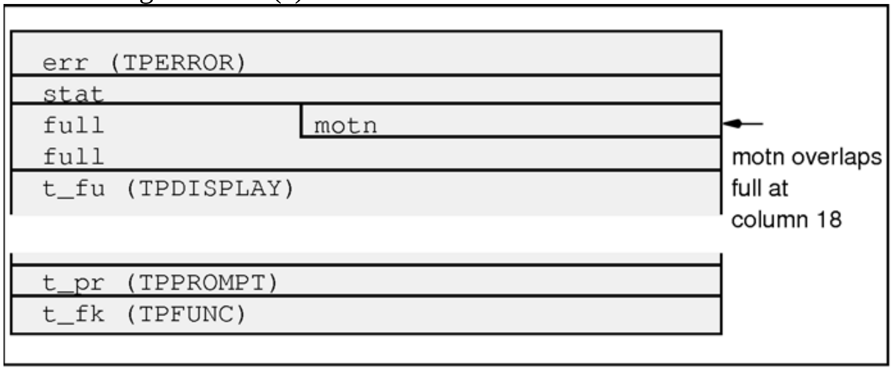


Fig. 7.10.1(a) "t_sc" Screen

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 7.10.1(b) .

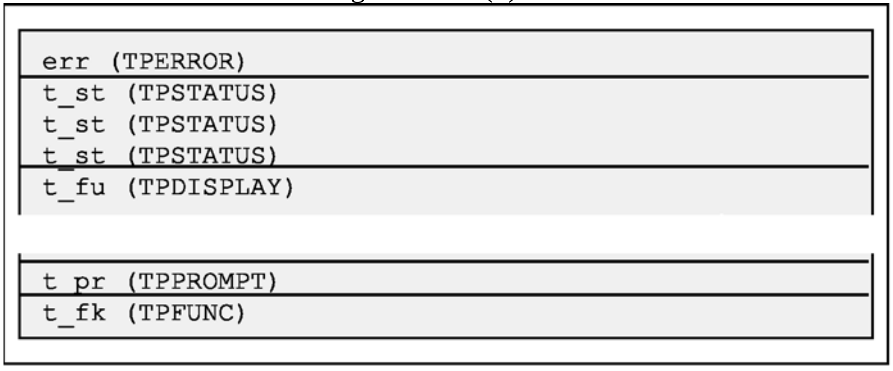


Fig. 7.10.1(b) "t_sc" Screen with $TP_USESTAT = TRUE

7.10.2 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 7.10.2 are defined for "c_sc".

Table 7.10.2 Defined Windows for c_sc"

Window Name	Lines	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
"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 an show FCTN and MENUS for F9 and F10. See Figure 7.10.2(a) .

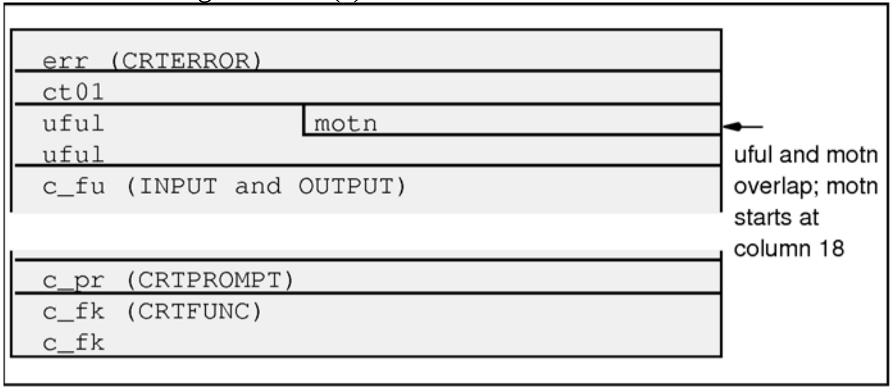
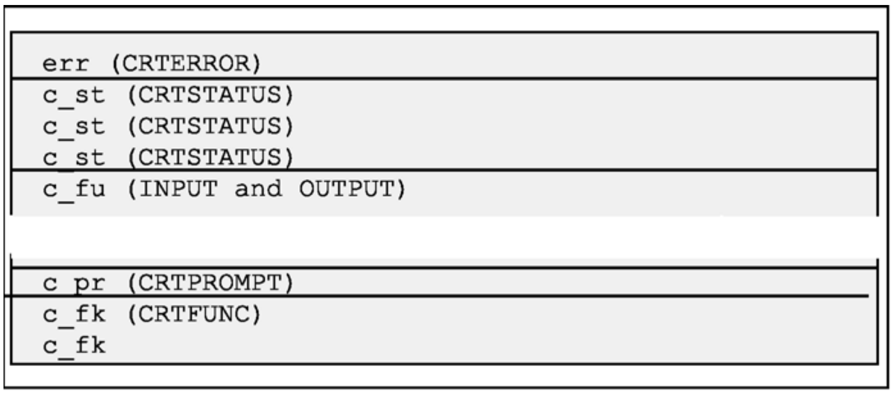


Fig. 7.10.2(a) "c_sc" Screen

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 7.10.2(b) .



```
err  (CRTERROR)
c_st (CRTSTATUS)
c_st (CRTSTATUS)
c_st (CRTSTATUS)
c_fu (INPUT and OUTPUT)

c_pr (CRTPROMPT)
c_fk (CRTFUNC)
c_fk
```

Fig. 7.10.2(b) "c_sc" Screen with $CRT_USERSTAT = TRUE

8 FILE SYSTEM

8.1 OVERVIEW

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 file system of KAREL allows you to use various devices. Refer to Subsection 8.2.1.

8.2 FILE SPECIFICATION

File specifications identify files. The specification indicates:

- The name of the device on which the file is stored, refer to Subsection 8.2.1 .
- The name of the file, refer to Subsection 8.2.2 .
- The type of data the file contains, refer to Subsection 8.2.3 .

The general form of a file specification is:

device_name:file_name.file_type

8.2.1 Device Name

A device name consists of at least two characters that indicate the device on which a file is stored. The device name always ends with a colon (:). The following is a list of valid storage devices. See the application-specific Operator's Manual about memory device.

- Memory card (MC:)
  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 slot is on the main board.
- FROM disk (FR:)
  Capable of storing program and other backups and any files. It can retain information when the power is interrupted, with no backup battery.
- RAM disk (RD:)
  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 the power is interrupted. It cannot be used with a spot tool.
- USB memory (UD1:)
  USB memory mounted to the USB port on the operator panel.
- USB memory (UT1:)
  USB memory mounted to the USB port on *i*Pendant.
  Option software A05B-2500-J957(USB port on *i*Pendant) is required to use UT1:.
- Memory device (MD:)
  The memory device is capable of handling data on the memory of a controller, such as robot programs and KAREL programs, as files.
- Memory device binary (MDB:)

The memory device binary (MDB:) is capable of saving backup as the same of FILE screen. It allows you to save backup remotely.

- Console (CONS:)
  Device for maintenance only. It can reference the log file containing internal information.
- MF disk (MF:)
  The MF disk is a device in which a FROM disk and a RAM disk are synthesized together. The file list on the MF disk displays the files on both the FROM 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.
- FTP (C1: to C8:)
  Writes and reads files to and from a 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.

8.2.2 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
```

8.2.3 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 8.2.3 is an alphabetical list of each available file type and its function.

Table 8.2.3 File type descriptions

File Type	Description
.BMP	**Bit map files** contain bit map images used in robot vision systems.
.CF	**KCL command files** are ASCII files that contain a sequence of KCL commands for a command procedure.
.DF	**Default file** are binary files that contain the default motion instructions for each teach pendant programming.
.DG	**Diagnostic files** are ASCII files that provide status or diagnostic information about various functions of the controller.
.DT	**KAREL data file** An ASCII or binary file that can contain any data that is needed by the user.
.IO	**Binary files** that contain I/O configuration data - generated when an I/O screen is displayed and the data is saved.
.KL	**KAREL source code files** are ASCII files that contain the KAREL language statements for a KAREL program.
.MN	**Mnemonic** program files are supported in previous version s of KAREL.
.ML	**Part model files** contain part model information used in robot vision systems.

File Type	Description
.PC	**KAREL p-code files** are binary files that contain the p-code produced by the translator upon translation of a .KL file.
.SV	**System files** 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	**Teach pendant program files** are binary files that contain instructions for teach pendant programs.
.TX	**Text files** are ASCII files that can contain system-defined text or user-defined text.
.VR	**Program variable files** are binary files that contain variable data for a KAREL program.
.VA	**ASCII variable files** are contain the listing of a variable file with variable names, types, and contents.
.LS	**Listing files** are teach pendant programs, error logs, and description histories in ASCII format.

8.3 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 FORMAT_DEV Built-in in Appendix A , "KAREL Language Alphabetical Description" or the FORMAT Command in Appendix C , "KCL Command Alphabetical Description."

8.3.1 Overview

The following kinds of storage devices can be used to store programs and files:

- Memory Card (MC:)
- Flash File Storage disk (FR:)
- RAM Disk (RD:) (Not for SpotTool+)
- FTP (C1:~C8:)
- Memory Device (MD:)
- Memory Device Binary (MDB:)
- MF Device (MF:)
- USB Memory Stick Device on the controller (UD1:)
- USB Memory Stick Device on the *i*Pendant (UT1:) (if you have an *i*Pendant with a USB port on it and you have the USB port on *i*Pendant option (J957) installed)

This Subsection 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 controller supports memory cards. Memory cards support various sizes 8MB or higher. 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.

⚠**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 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 device such as ATA Flash PC card.

USB Memory Stick Device on the controller (UD1:), and USB Memory Stick Device on the *i*Pendant (UT1:)

The controller supports USB Flash memory. It allows you to load or save files. Refer to the application-specific Operator's Manual.

NOTE

In order to use the UT: device on the *i*Pendant, you must have the USB Port on *i*Pendant option (J957) installed.

NOTE

Some USB memory products cannot be recognized correctly by the robot controller or accept file operations correctly.

Those USB memory units that provide secure functions and require password authentication before access to the drive cannot be used.

Write protect notch of USB memory may not be functional.

Those USB memory units that have been confirmed for operation are as follows:

Clip Drive RUF-CL/U2 128/256/512MB/1GB of BUFFALO INC.
ToteBag TB-BH 512MB/1GB of I-O DATA DEVICE INC.

Software vesion must be newer than V7.20P/43 V7.30P/31 V7.40P/13 V7.50P/01 to use 4GB(or more) USB memory
Those USB memory units that have been confirmed for operation (at software version newer than V7.20P/43 V7.30P/31 V7.40P/13 V7.50P/01) are as follows:

RUF2 16GB Flash Drive RUF2 4GB RUF 8GB of BUFFALO INC.
ToteBag TB-BH2 16GB TB-ST 8GB of I-O DATA DEVICE INC.
Cruzer Contour 16GB ruzer Colors+ 8GB of SanDisk INC.

NOTE

1 Those USB memory units above are confirmed for operation, but FANUC does not guarantee about USB memory on the market and does not accept responsibility to defective unit or malfunction by specification changes of device. Please confirm a USB memory unit at your site before you use it.
2 Robot controller cannot format some USB memory product. In this case please use PC to format it.
3 Necessary time to ready differ depending on the USB memory product. Especially the USB memory formatted by FAT32 needs long time to ready. In this case, format it by FAT on PC.

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

You can format the Flash file storage disk. The size of the Flash file storage disk is set by the system at software installation.

RAM Disk (Not for SpotTool+)

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 disks 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 magnetic disks 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 (C1: ~ C8:)

Writes and reads files to and from a 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 “FANUC Robot series R-30*i*A/R-30*i*A Mate CONTROLLER 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)

Memory Device Binary (MDB:)

The memory device binary (MDB:) device 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:).

MF Device (MF:)

MF: is a composite device that will search the RAM Disk (RD:) and flash file storage disk (FR:) devices, in that order, for a specified file. MF: eliminates your need to know the name of the device that contains the file you specify. For example, "DIR MF:file.ext" will search for the file first on RD:. If it is not found, it will search for the file on FR:. Also, "COPY MC:file.ext to MF" will place the file on RD:.
When files are copied to the MF: device, the RAM Disk is used by default if RD: is in SRAM($FILE_MAXSEC > 0). The Flash ROM disk is used by default if RD: is in DRAM ($FILE_MAXSEC < 0).

NOTE
When you are backing up files, the MF: device will prompt you to select either FR: or the RD: device. The files will be copied to the device that you selected even if RD: is in DRAM.

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. Refer to the application-specific Operator's Manual for more information.

8.3.2 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 the appropriate application-specific Operator's Manual.
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.
For more information on memory, refer to Subsection 1.4.1 .

8.4 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 OS.

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 builtin. By default, the read operation is not in snapshot mode.

Typical pipe applications involve one process writing data to a pipe. The data can 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 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 8.4(a) have the following meanings:

Table 8.4 (a) 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.17 , "pipe_config built-in" 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.

Example 8.4 (b) Program

```
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_atr(att_file, ATR_IA)   --force reads to completion
 set_file_atr(att_file, ATR_FIELD)  --force write to completion
 set_file_atr(att_file, ATR_PASSALL) --ignore cr
 set_file_atr(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_ATR(console, atr_ia, 0) --ATR_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
```

8.5 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 8.5(a) for listings and descriptions of files available on the MD device.

Table 8.5 (a) File listings for the MD device

File Name	Description
ACCNTG.DG	This file shows the system accounting of Operating system tasks.
AXIS.DG	This file shows the Axis and Servo Status.
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.

File Name	Description
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.DG	This file shows the Ethernet Configuration.
FRAME.DG	This file shows Frame assignments.
FRAMEVAR.SV	This file contains system frame and tool variable information in binary form.
FRAMEVAR.VA	This file is an ASCII listing of FRAMEVAR.SV.
HIST.LS	This file shows history register dumps.
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.
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.
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 *Internet Options Manual* for more information .
RIPESTAT.DG	This file contains performance data for you to determine how well the network is performing. Refer to the *Internet Options Manual* for more information .
SFTYSIG.DG	This file is an ASCII listing of the state of the safety signals.
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.

File Name	Description
SYSSERVO.VA	This file is an ASCII listing of SYSSERVO.SV.
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.
***.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 *i*Pendant Control or Server Side Include.
***.GIF	This file is a GIF image file.
***.JPG	This file is a JPEG image file.

Refer to Table 8.5(b) for a listing of restrictions when using the MD: device.

Table 8.5 (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	
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	

9 DICTIONARIES AND FORMS

9.1 OVERVIEW

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 9.2 .
- Creating and using forms, refer to Section 9.3 .

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.

9.2 CREATING USER DICTIONARIES

A dictionary file provides a method for customizing displayed text, including the text attributes (blinking, underline, double wide, etc.), and the text location on the screen, without having to re-translate the program.

⚠CAUTION
Create dictionaries by ROBOGUIDE basically. Refer to Appendix F for more information.

The following are steps for using dictionaries. It allows to use KCL too.

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 Subsection 10.2.12 , "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.

9.2.1 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>
$n<,ele_name><@cursor_pos><&res_word><#chr_code><"Ele_text"><&res_wor d>
<#chr_code><+nest_ele>
<*comment>
<$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 Subsection 10.2.9 .
- $n specifies the element number. n is a positive integer 0 or greater. Refer to Subsection 10.2.2 .
- ,ele_name specifies a comma followed by the element name. Refer to Subsection 10.2.3 .
- @cursor_pos specifies a cursor position (two integers separated by a comma.) Cursor positions begin at @1,1. Refer to Subsection 10.2.4 .
- &res_word specifies a dictionary reserve word. Refer to Subsection 10.2.6 .
- "Ele_text" specifies the element text to be displayed. Refer to Subsection 10.2.5 .
- +nest_ele specifies the next dictionary text. Refer to Subsection 10.2.8 .

- 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 10.2.6 lists the attributes of a dictionary element.

9.2.2 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
  $-

9.2.3 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.
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.

9.2.4 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 Subsection 7.10.1 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.
  - The lower right position of the window is "@17,80" and is located at the "c_sc" screen row 21, column 80.

Refer to Subsection 7.10.2 for more information on the CRT/KB screens and windows.
The window size defines the display limits of the dictionary elements.

9.2.5 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 Subsection 10.2.6 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 9.2.5 .
- ^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 9.2.5 lists the conversion characters:

Table 9.2.5 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.dddddd, 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.dddddd, 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.

9.2.6 Dictionary Reserved Word Commands

Reserve words begin with the ``&" character and are used to control the screen. They effect 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 9.2.6 for a list of the available reserved words.

Table 9.2.6 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)
&new_line	New line (#135)

Reserved Word	Function
&bs	Back space (#136)
&home	Home cursor in window (#137)
&reverse	Reverse video attribute (#139)
&standard	All attributes normal (#143)

9.2.7 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 , for a listing of the ASCII character codes.

9.2.8 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.

9.2.9 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.

9.2.10 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 fie. 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.

9.2.11 Compressing and Loading Dictionaries on the Controller

⚠CAUTION

Create dictionaries by ROBOGUIDE basically. Refer to Appendix F for more information.

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 9.2.11 illustrates the compression process.

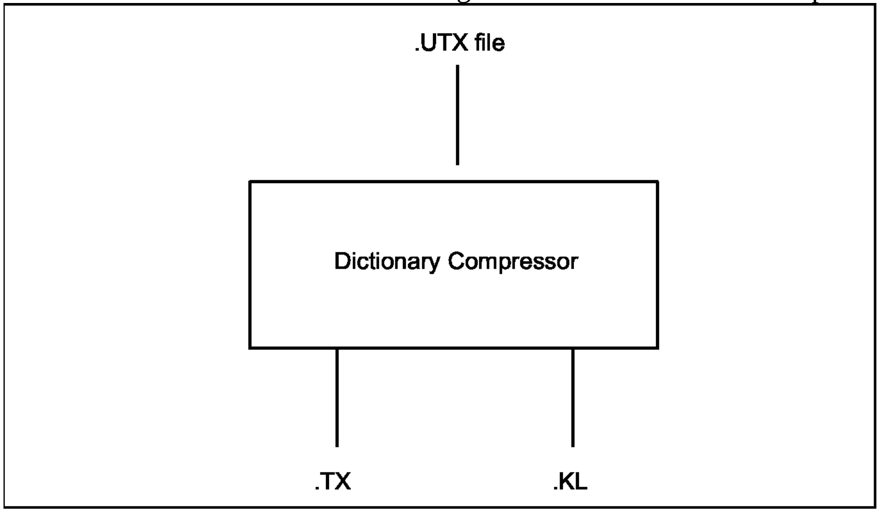


Fig. 9.2.11 Dictionary compressor and user dictionary file

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.

9.2.12 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.

9.3 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 R-30*i*A menu interface.

⚠**CAUTION**
Create dictionaries by ROBOGUIDE basically. Refer to Appendix F for more information.

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 DISCTRL_FORM built-in, 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.

9.3.1 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.
The following is the syntax for a form:

Example 9.3.1 Form Syntax

```
.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>
    <"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">
```

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 *i*Pendant will display the color. The monochrome pendant will ignore the color attributes.

9.3.2 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
```

9.3.3 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 R-30*i*A 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
```

9.3.4 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 Subsections.

9.3.5 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.
- 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 " &new_line
```

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 of DISCTRL_FORM. 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.

9.3.6 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 9.3.6 lists the conversion characters for an editable data item.

Table 9.3.6 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.

Character	Argument Type: Printed As
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.dddddd, 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.dddddde+-xx or <->m.ddddddE+-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.
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 %.

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.

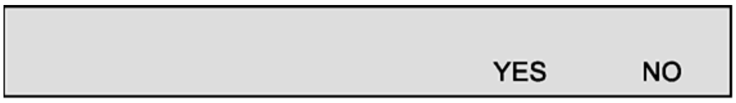
Edit Data Item: BOOLEAN

- 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"
```



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.

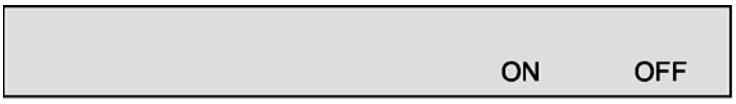
Edit Data Item: BOOLEAN Port

- 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"
```



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,

```
"-%12pk(1)" * specifies TP programs

"-%12pk(2)" * specifies PC programs

"-%12pk(6)" * specifies TP, PC, VR

"-%12pk(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)"
```

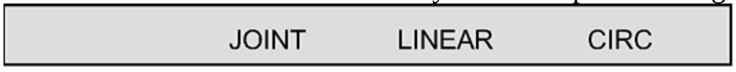
The dictionary element defining the function keys should list one function key label per line. If function keys to the left of those specified are not active, they should be set to "". A maximum of 5 function keys can be used. For example,

```
$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.



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 must be defined as a KAREL string array. Each element of the array should define a choice in the subwindow. This format specifier is similar to %w. However, the first element is related to the value 0 and is never used. Value 1 begins at the second element. The last value is either the end of the array or the first uninitialized value.

```
[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*
```

9.3.7 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.

9.3.8 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 Subsection for an edit data item.

9.3.9 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.

9.3.10 Form Reserved Words and Character Codes

Reserved words or character codes can be used. Refer to Table 9.3.10(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 Table 9.3.10(b) for a list of available reserved words for a scrolling window.

Table 9.3.10(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)
&new_line	New line (#135)
&bs	Back space (#136)
&home	Home cursor in window (#137)
&reverse	Reverse video attribute (#139)
&standard	All attributes normal (#143)

Table 9.3.10(b) lists the reserved words that can be used for a scrolling window.

Table 9.3.10(b) Reserved words for scrolling window

Reserved Word	Function
&new_line	New line (#135)
&reverse	Reverse video attribute (#139)
&standard	All attributes normal (#143)

9.3.11 Form Function Key Element Name or Number

Each form can have one related function key menu. A function key menu has the following characteristics:

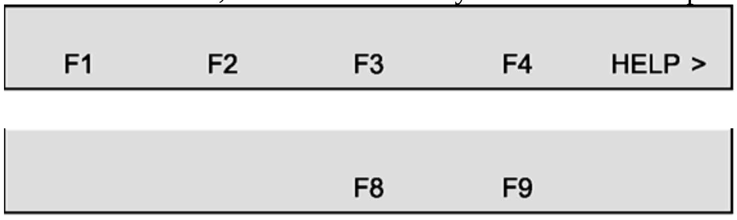
- The function key menu is specified in the dictionary with a caret, ^, immediately followed by the name or number of the function key dictionary element. For example,

```
^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.



9.3.12 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] ‘       >’
```

9.3.13 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.

9.3.14 Teach Pendant Form Screen

You can write to other active teach pendant windows while the form is displayed. The screen itself is named "tpsc." Figure 9.3.14 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.

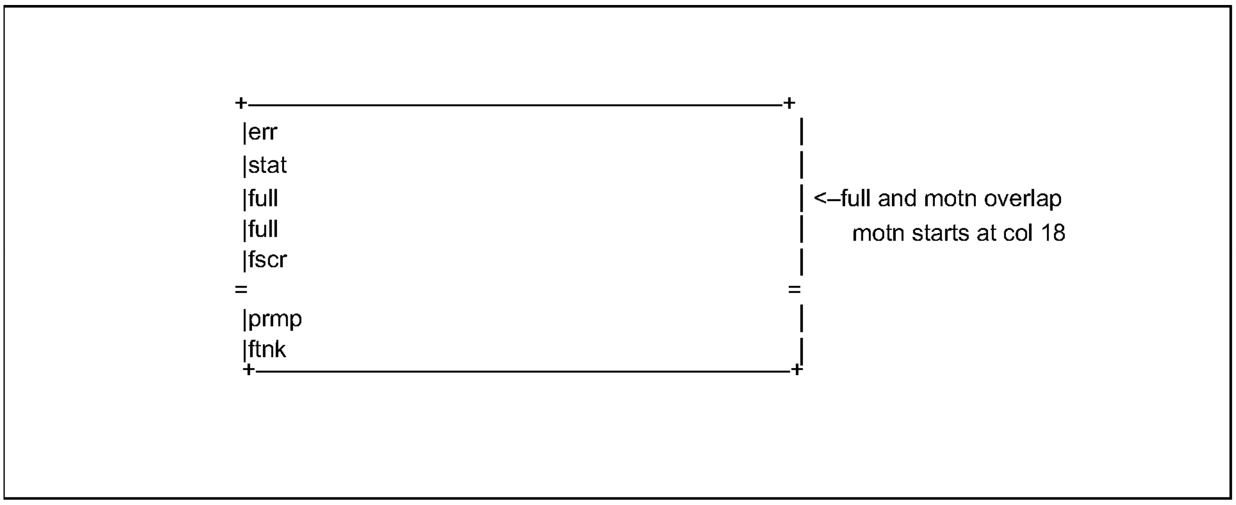


Fig. 9.3.14 Teach pendant form screen

9.3.15 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 9.3.15 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.

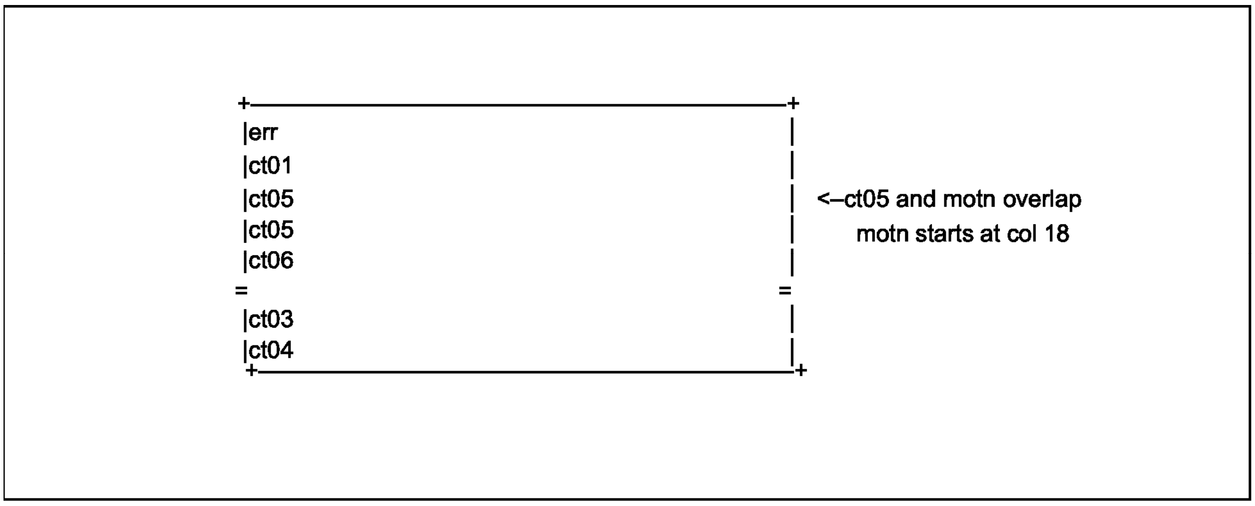


Fig. 9.3.15 CRT/KB form screen

9.3.16 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.

9.3.17 Compressing and Loading Forms on the Controller

⚠CAUTION
Compress forms by ROBOGUIDE basically. Refer to Appendix F for more information.

Compressing with KCL is as follows.
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.

NOTE
The form file must be an uncompressed file in order for the errors to point to the correct line.

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 9.3.17 illustrates compressing.

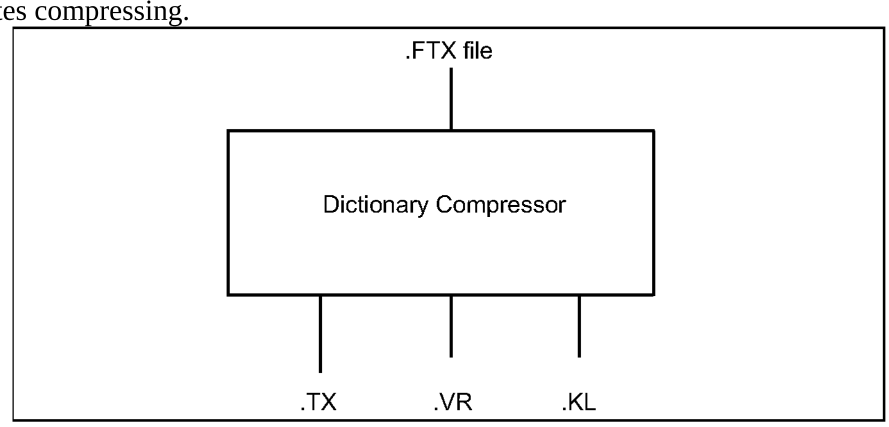


Fig. 9.3.17 Dictionary compressor and form dictionary file

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
```

9.3.18 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                      5/5
  1  Menu item 1
  2  Menu item 2
  3  Menu item 3
  4  Menu item 4
  5  Menu item 5
```

Fig. 9.3.18(a) Example of selectable menu items

The dictionary elements in FORM.FTX, shown in Example 9.3.18(a), were used to create the form shown in Figure 9.3.18(a) .

Example 9.3.18 (a) Example form dictionary for selectable menu items

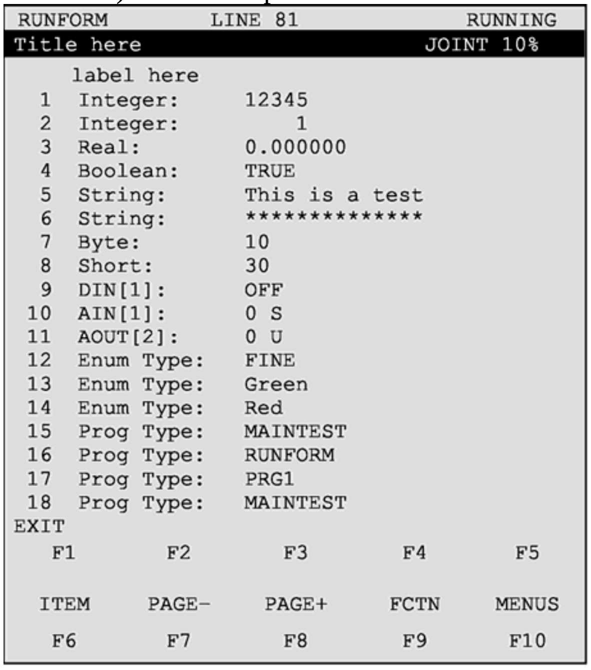
```
*     Dictionary Form File: form.ftx
*
*     Generate form.kl which should be included in your KAREL program
.kl form
.form
$-,forml
     &home &reverse "Title here" &standard $new_line
     "   label here " &new_line
     @3,10   "- Menu item 1 "
     @4,10   "- Menu item 2 "
     @5,10   "- Menu 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
```

The program shown in Example 9.3.18(b) was used to display the form shown in Figure 9.3.18(a) .

Example 9.3.18(b) Example program for selectable menu items

```
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 9.3.18(b) 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%
     label here
  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
```

Fig. 9.3.18(b) Example of edit data items

The dictionary elements in FORM.FTX, shown in Example 9.3.18(c) , were used to create the form shown in Figure 9.3.18(b) .

Example 9.3.18(c) Example dictionary for edit data items

```
* 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
"   Bolean:       "   "-%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
"VARDECEL" &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
```

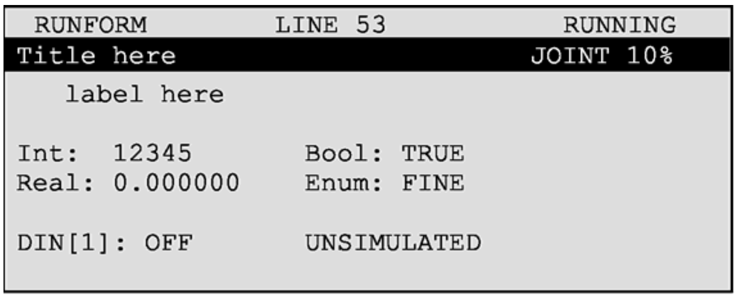
The program shown in Example 9.3.18(d) was used to display the form in Figure 9.3.18(b) .

Example 9.3.18(d) Example program for edit data items

```
PROGRAM runform2
%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*]$group[1].$termtype'
     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 runform2
```

Figure 9.3.18(c) 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.



```
 RUNFORM         LINE 53          RUNNING
Title here                      JOINT 10%
   label here

Int:  12345       Bool: TRUE
Real: 0.000000    Enum: FINE

DIN[1]: OFF       UNSIMULATED
```

Fig. 9.3.18(c) Example of Display Only Data Items

The dictionary elements in FORM.FTX, shown in Example 9.6 , were used to create the form shown in Figure 9.3.18(e) .

Example 9.3.18(e) Example dictionary for display only data items

```
*      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
```

The program shown in Example 9.3.18(f) was used to display the form shown in Figure 9.3.18(c) .

Example 9.3.18 (f) Example program for display only data items

```
PROGRAM runform3
%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*]$group[1].$termtype'
     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 runform3
```

10 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.

10.1 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, 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 8 "FILE SYSTEM".

10.1.1 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

10.1.2 Variables and Data Types

The KCL> SET VARIABLE command permits you to assign values to declared variables. Assigned values can be INTEGER, REAL, BOOLEAN, and STRING data types. VECTOR variables are assigned as three REAL values, and POSITION variables are assigned as six REAL values.
See Also: SET VARIABLE KCL commands in Appendix C , “KCL Command Alphabetical Description”

10.2 ENTERING COMMANDS

You can enter KCL commands only from the CRT/KB.
To enter KCL commands:

1. Press MENUS (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.

10.2.1 Abbreviations

Any KCL command can be abbreviated as long as the abbreviations are unique in KCL. For example, **TRAN** is unique to TRANSLATE.

10.2.2 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.

10.2.3 Subdirectories

Subdirectories are available on the memory card device. Subdirectories allow both memory cards and Flash disk cards. You can nest subdirectories up to many levels. However, It is not recommended to nest subdirectories greater than eight levels.

10.3 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.

10.3.1 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. Example 10.1 displays **CLEAR PROGRAM AA_01** as the KCL command and **YES** as the confirmation.

Example 10.3.1 Confirmation in a command procedure

```
CLEAR PROGRAM AA_01 YES
```

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 Subsection 13.4.3 for more information on $STOP_ON_ERR.

See Also: Subsection 10.3.3 , “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:

```
SET VARIABLE &
[CFAMP01]int_var = 1
```

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.

10.3.2 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: Appendix C , “KCL Command Alphabetical Description”

10.3.3 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.

10.3.4 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.
Command procedures can be executed using the KCL RUNCF command.

11 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.

11.1 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 OPEN HAND and CLOSE HAND statements.

11.1.1 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.

11.1.2 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.

11.1.3 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
```

Fig. 11.1.3 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.

11.1.4 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 RO[1] and RO[2].
HAND[2] uses the same physical outputs as RO[3] and RO[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

11.2 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 (RI) and robot digital output (RO)
- Operator panel input (OPIN) and operator panel output (OPOUT)
- Teach pendant input (TPIN) and teach pendant output (TPOUT)

11.2.1 Robot Digital Input and Output Signals (RI/RO)

Robot I/O is the input and output signals between the controller and the robot. These signals are sent to the EE (End Effecter) connector located on the robot. The number of robot input and output signals (RI and RO) varies depending on the number of axes in the system. For more information on configuring Robot I/O, refer to the appropriate application-specific Operator's Manual.
RI[1] through RI[8] are available for tool inputs. All or some of these signals can be used, depending on the robot model. Refer to the Mechanical Unit Manual specific to your robot model, for more information.
RO[1] through RO[8] are available for tool control. All or some of these signals can be used, depending on the robot model. Refer to the Mechanical Unit Manual specific to your robot model, for more information.
RO[1] through RO[4] are the same signals set using OPEN, CLOSE, and RELAX hand. See Subsection 11.1.4 .

11.2.2 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 11.2.2(a) lists each standard operator panel input signal. Table 11.2.2(b) lists each standard operator panel output signal.

Table 11.2.2(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 11.2.2(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 11.2.2(c) lists and describes each user operator panel input signal. Table 11.2.2(d) lists each user operator panel output signal. Figure 11.2.2(a) and Figure 11.2.2(b) illustrate the timing of the UOP signals. Refer to application-specific operator's manual for more information.

Table 11.2.2(c) User operator panel input signals

OPIN[n]	UI[n]	Function	Description
OPIN[16]	UI[1]	*IMSTP Always active	The *IMSTP input is on in the normal status. When this signal is turned off, the following processing is performed: ● An alarm is generated and the servo power is turned off. ● The robot operation is stopped immediately. Execution of the program is also stopped ⚠**WARNING** *IMSTP is a software controlled input and cannot be used for safety purposes. Use *IMSTP with EMG1 and EMG2 to use this signal with a hardware controller emergency stop. Refer to the Mechanical Unit Manual, specific to your robot model, for connection information of EMG1 and EMG2.
OPIN[17]	UI[2]	*HOLD Always active	*HOLD is the external hold signal. The *HOLD input is on in the normal status. When this signal is turned off, the following processing is performed: ● The robot is decelerated until its stops, then the program execution is halted. ● If ENABLED is specified at ”Break on hold” on the general item setting screen, the robot is stopped, an alarm is generated, and the servo power is turned off.

OPIN[n]	UI[n]	Function	Description
OPIN[18]	UI[3]	*SFSPD Always active	The safety speed signal temporarily stops the robot when the safety fence door is opened. This signal is normally connected to the safety plug of the safety fence door. The *SFSPD input is on in the normal status. When this signal is turned off, the following processing is performed: ● The operation being executed is decelerated and stopped, and execution of the program is also stopped. At this time, the federate override is reduced to the value specified for $SCR.$FENCEOVRD. ● When the *SFSPD input is off and a program is started from the teach pendant, the federate override is reduced to the value specified for $SCR.$SFRUNOVLIM. When jog feed is executed, the federate override is reduced to the value specified for $SCR.$SFJOGOVLIM. When *SFSPD is off, the federate override cannot exceed these values. ⚠WARNING *SFSPD is a software controlled input and cannot be used for safety purposes. Use * SFSPD with FENCE1 and FENCE2 to use this signal with a hardware controller emergency stop. Refer to the Mechanical Unit Manual, specific to your robot model, for connection information of FENCE1 and FENCE2.
OPIN[19]	UI[4]	CSTOPI Always active	The cycle stop signal terminates the program currently being executed. It also releases programs from the wait state by RSR. ● When FALSE is selected for CSTOPI for ABORT on the Config system setting screen, this signal terminates the program currently being executed as soon as execution of the program completes. It also releases (Clear) programs from the wait state by RSR. (Default) ● When TRUE is selected for CSTOPI for ABORT on the Config system setting screen, this signal immediately terminates the program currently being executed. It also releases (Clear) programs from the wait state by RSR. ⚠ WARNING When FALSE is selected for CSTOPI for ABORT on the Config system setting screen, CSTOPI does not stop the program being executed until the execution is complete.
OPIN[20]	UI[5]	FAULT_RESET Always active	FALT_RESET is the external fault reset signal. If the servo power is off, the RESET signal turns on the servo power. The alarm output is not canceled until the servo power is turned on. The alarm is canceled at the instant this signal falls in default setting.

OPIN[n]	UI[n]	Function	Description
OPIN[21]	UI[6]	START Active when the robot is in a remote condition (CMDENBL = ON)	This is an external start signal. This signal functions at its falling edge when turned off after being turned on. When this signal is received, the following processing is performed: ● When FALSE is selected for START for CONTINUE only on the Config system setting screen, the program selected using the teach pendant is executed from the line to which the cursor is positioned. A temporarily stopped program is also continued. (Default) ● When TRUE is selected for START for CONTINUE only on the Config system setting screen, a temporarily stopped program is continued. When the program is not temporarily stopped, it cannot be started.
OPIN[23]	UI[8]	ENBL Always active	The ENBL signal allows the robot to be moved and places the robot in the ready state. When the ENBL signal is off, the system inhibits a jog feed of the robot and activation of a program including a motion (group). A program which is being executed is halted when the ENBL signal is set off.
OPIN[24] - OPIN[31]	UI[9]- UI[16]	RSR1/PNS1, RSR2/PNS2, RSR3/PNS3, RSR4/PNS4,RSR5/PNS5, ,RSR6/PNS6, , RSR7/PNS7, ,RSR8/PNS8 Active when the robot is in a remote condition (CMDENBL = ON)	RSR1-8 are robot service request signals. When one of these signals is received, the RSR program corresponding to the signal is selected and started to perform automatic operation. When another program is being executed or is stopped temporarily, the selected program is added to the queue and is started once the program being executed terminates. PNS1-8 are program number select signals and a PN strobe signal. When the PNSTROBE input is received, the PNS1 to PNS8 inputs are read to select a program to be executed. When another program is being executed or temporarily stopped, these signals are ignored. When the remote conditions are satisfied, program selection using the teach pendant is disabled while PNSTROBE is on.
OPIN[32]	UI[17]	PNSTROBE Active when the robot is in a remote condition (CMDENBL = ON)	The PNSTROBE is "Program Number Select Strobe". Refer to the description of PNS.
OPIN[33]	UI[18]	PROD_START Active when the robot is in a remote condition (CMDENBL = ON)	The automatic operation start (production start) signal starts the currently selected program from line 1. This signal functions at its falling edge when turned off after being turned on. When this signal is used together with a PNS signal, it executes the program selected by the PNS signal starting from line 1. When this signal is used together with no PNS signal, it executes the program selected using the teach pendant starting from line 1. When another program is being executed or temporarily stopped, this signal is ignored. See Figure 11.2.2(b) .

Table 11.2.2(d) User operator panel output signals

OPOUT[n]	UO[n]	Function	Description
OPOUT[16]	UO[1]	CMDENBL	The input accept enable (command enable) signal is output when the following conditions are satisfied. This signal indicates that a program including an operation (group) can be started from the remote controllers. ● The remote conditions are satisfied. ● The operation enable conditions are satisfied. ● The mode is continuous operation (single step disable).
OPOUT[17]	UO[2]	SYSRDY	SYSRDY is output while the servo power is on. This signal places the robot in the operation enable state. In the operation enable state, jog feed can be executed and a program involving an operation (group) can be started. The robot enters the operation enable state when the following operation enable conditions are satisfied: ● The ENBL input of the peripheral device I/O is on. ● The servo power is on (not in the alarm state).
OPOUT[18]	UO[3]	PROGRUN	PROGRUN is the program run output. This output turns on when a program is running. It is not output while a program is temporarily stopped. See Figure 11.2.2(b) .
OPOUT[19]	UO[4]	PAUSED	PAUSED is the paused program output. This output turns on when a program is paused and waits for restart.
OPOUT[20]	UO[5]	HELD	HELD is output when the hold button is pressed or the HOLD signal is input. It is not output when the hold button is released.
OPOUT[21]	UO[6]	FAULT	FAULT is output when an alarm occurs in the system. The alarm state is released by the FAULT_RESET input. FAULT is not output when a warning (WARN alarm) occurs.
OPOUT[22]	UO[7]	ATPERCH	ATPERCH is output when the robot is in a previously defined reference position. Up to three reference positions can be defined. This signal is output only when the robot is in the first reference position. For any other reference positions, general-purpose signals are assigned. See application-specific Operator's Manual of Section "SETTING A REFERENCE POSITION".
OPOUT[23]	UO[8]	TPENBL	TPENBL is output when the enable switch of the teach pendant is set to on.
OPOUT[24]	UO[9]	BATALM	BATALM indicates a low-voltage alarm for the backup battery of the controller or robot pulsecoder. Turn the power to the controller on and replace the battery.
OPOUT[25]	UO[10]	BUSY	BUSY is output while a program is being executed or while processing using the teach pendant is being performed. It is not output while a program is temporarily stopped.

OPOUT[n]	UO[n]	Function	Description
OPOUT[26] OPOUT[33]	UO[11]- UO[18]	ACK1/SNO1, ACK2/SNO2, ACK3/SNO3, ACK4/SNO4, ACK5/SNO5, ACK6/SNO6, ACK7/SNO7, ACK8/SNO8	**ACK 1-8** are the acknowledge signals output 1 through 8. When the RSR function is enabled, ACK1 to ACK8 are used together with the function. When an RSR input is accepted, a pulse of the corresponding signal is output as an acknowledgment. The pulse width can be specified. See application-specific Operator's Manual for more information about RSR. See Figure 11.2.2(a) **SNO 1-8** are the signal number outputs. When the PNS function is enabled, SNO1 to SNO8 are used together with the function. The currently selected program number (signal corresponding to the PNS1 to PNS8 inputs) is always output, in binary code, as confirmation. The selection of another program changes SNO1 to SNO8. See application-specific Operator's Manual for more information about RSR. See Figure 11.2.2(b) .
OPOUT[34]	UO[19]	SNACK	SNACK is the signal number acknowledge output. When the PNS function is enabled, SNACK is used together with the function. When the PNS inputs are accepted, a pulse of this signal is output as an acknowledgment. The pulse width can be specified. See Figure 11.2.2(b) .

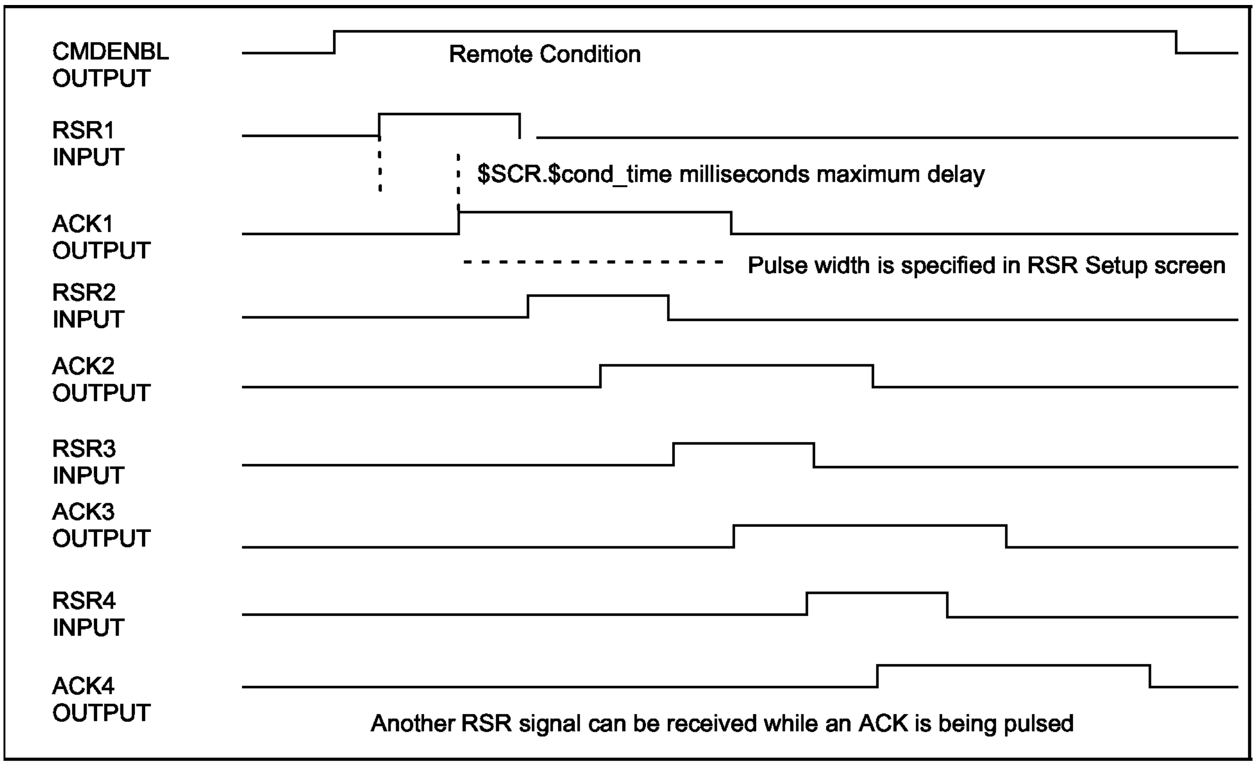

CMDENBL OUTPUT
Remote Condition
RSR1 INPUT
$SCR.$cond_time milliseconds maximum delay
ACK1 OUTPUT
Pulse width is specified in RSR Setup screen
RSR2 INPUT
ACK2 OUTPUT
RSR3 INPUT
ACK3 OUTPUT
RSR4 INPUT
ACK4 OUTPUT
Another RSR signal can be received while an ACK is being pulsed

Fig. 11.2.2(a) RSR timing diagram

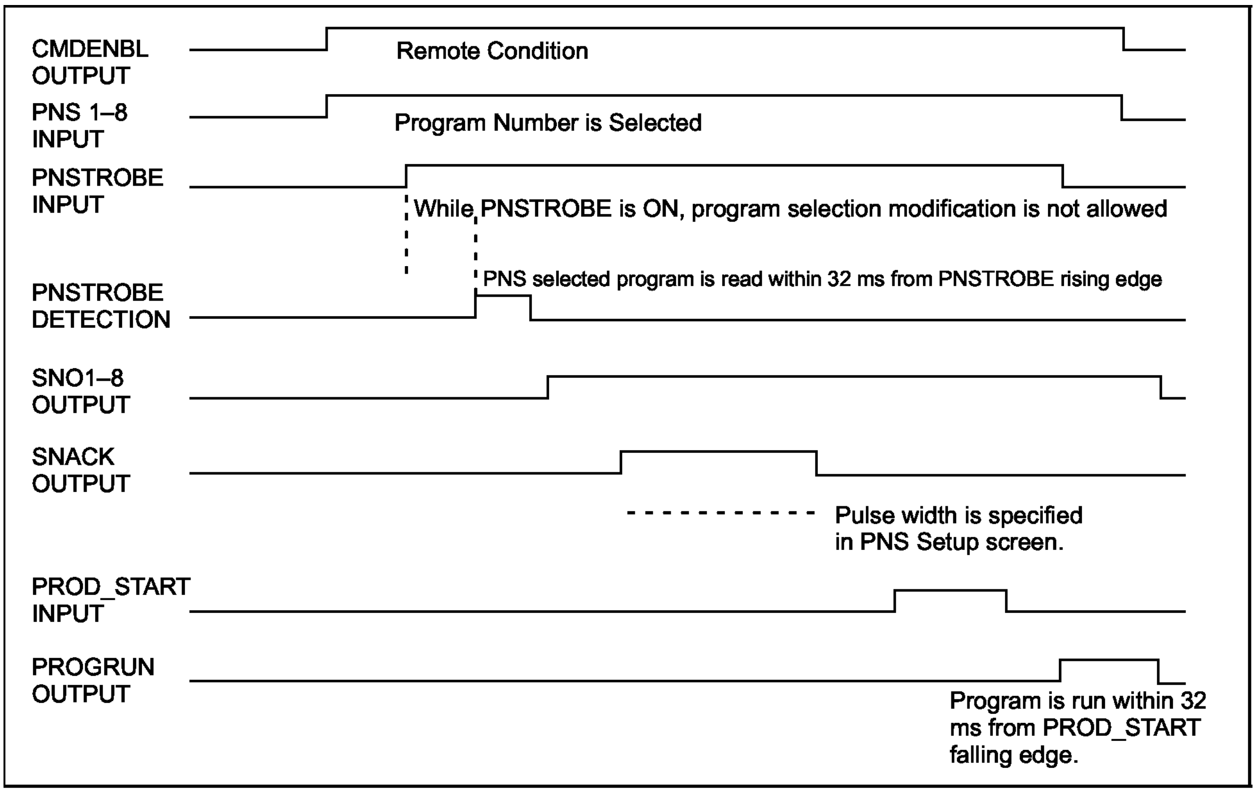

CMDENBL OUTPUT
Remote Condition
PNS 1–8 INPUT
Program Number is Selected
PNSTROBE INPUT
While PNSTROBE is ON, program selection modification is not allowed
PNS selected program is read within 32 ms from PNSTROBE rising edge
PNSTROBE DETECTION
SNO1–8 OUTPUT
SNACK OUTPUT
Pulse width is specified in PNS Setup screen.
PROD_START INPUT
PROGRUN OUTPUT
Program is run within 32 ms from PROD_START falling edge.

Fig. 11.2.2(b) PNS timing diagram

11.2.3 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 11.2.3 for teach pendant input signal assignments.

Table 11.2.3 Teach pendant input signal assignments

TPIN[n]	Teach Pendant Key
EMERGENCY STOP AND DEADMAN	
TPIN[250]	EMERGENCY STOP
TPIN[249]	ON/OFF switch
Arrow Keys	
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
Keypad Keys (shifted or unshifted)	

TPIN[n]	Teach Pendant Key
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
Function Keys	
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
Menu Keys	
TPIN[143]	SELECT
TPIN[144]	MENUS
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
User Function Keys	

TPIN[n]	Teach Pendant Key
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
Motion Keys	
TPIN[185]	FWD
TPIN[186]	BWD
TPIN[187]	COORD
TPIN[188]	+X
TPIN[189]	+Y
TPIN[190]	+Z
TPIN[191]	+X rotation
TPIN[192]	+Y rotation
TPIN[193]	+Z rotation
TPIN[194]	-X
TPIN[195]	-Y
TPIN[196]	-Z
TPIN[197]	-X rotation
TPIN[198]	-Y rotation
TPIN[199]	-Z rotation
TPIN[226]	Shifted FWD
TPIN[207]	Shifted BWD
TPIN[202]	Shifted COORD
Motion Keys Cont'd	
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[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

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

11.3 SERIAL INPUT/OUTPUT

11.3.1 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 Mechanical Unit 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.
Figure 11.3.1 shows the location of the ports on the controller.

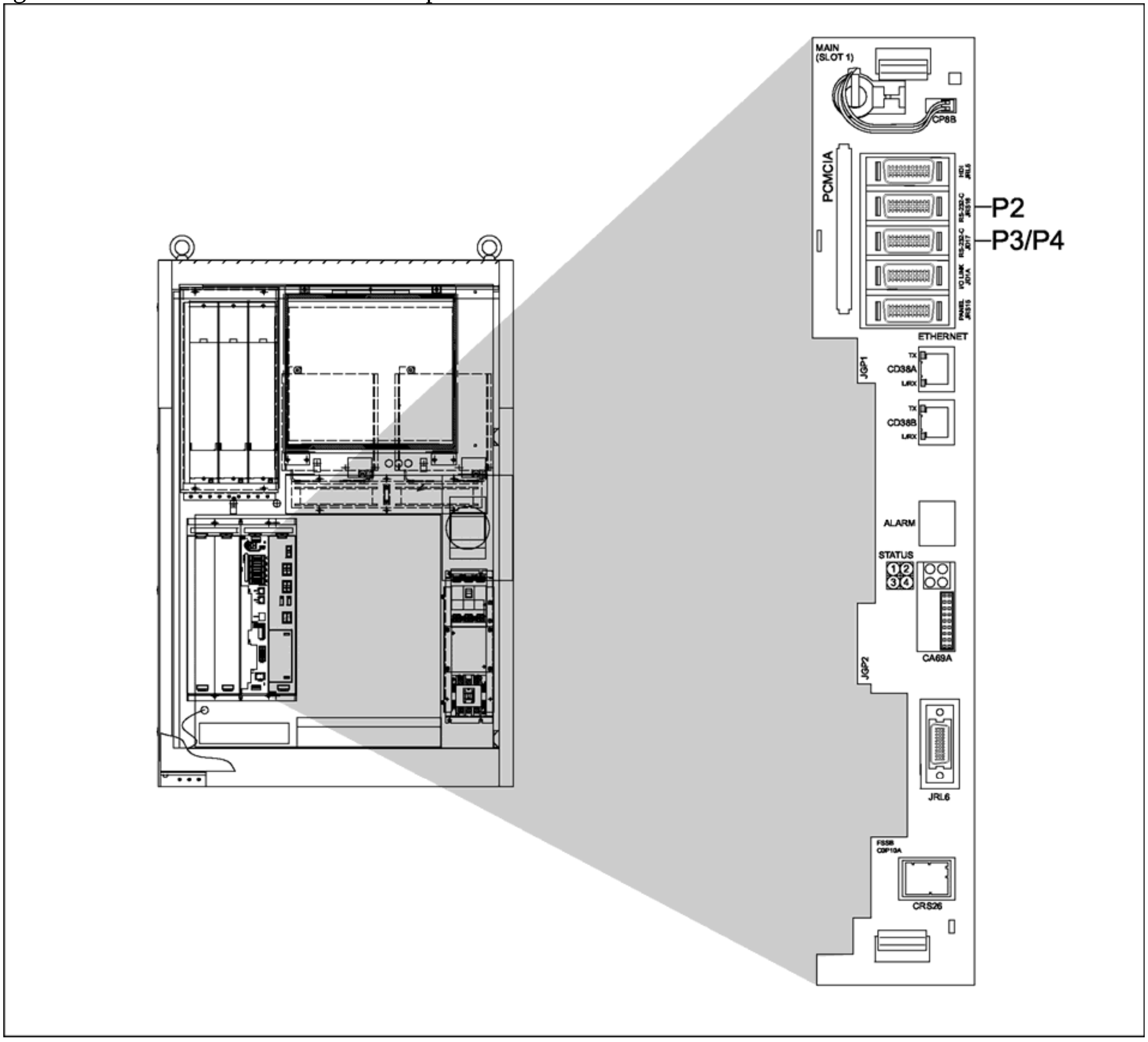


Fig. 11.3.1 Location of ports on the controller

Ports

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.

Up to four ports are available, P1-P4. Table 11.3.1(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.

Table 11.3.1(a) Ports P1 - P4

Port	Item Name on Screen	Kind of Port	Use	Default Device
P1	Teach Pendant **NOTE** 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 11.3.1(b) lists the default settings for each kind of device you can connect to a port.

Table 11.3.1(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
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 A for more information on the statements and built-ins available in KAREL
Refer to the application-specific Operator's Manual for more information about setting up ports.

12 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 12.3, “Task Scheduling” for information on how the system decides which task to execute first.

12.1 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.

12.2 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.

12.3 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 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 Subsection 12.4.1 , “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 Subsection 12.4.2 , “Time Slicing,” for more information.

12.3.1 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 be 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 12.3.1 System function priority table

Priority	System Function	Effect of Delaying Function
-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.

Priority	System Function	Effect of Delaying Function
143	Lowest priority	Does not delay any of the above.

12.3.2 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_TSK_ATTR built-in.

12.4 STARTING TASKS

There are a number ways to start a task.

- KCL RUN command. Refer to Appendix C ,"KCL Command Alphabetic Descriptions."
- Operator Panel start key. Refer to the appropriate application-specific Operator's Manual.
- User operator panel start signal. Refer to the appropriate application-specific Operator's Manual.
- Teach pendant shift-FWD key. Refer to the appropriate application-specific Operator's Manual, Chapter on "Testing a Program and Running Production," for more information.
- Teach pendant program executes a RUN instruction. Refer to Subsection 12.4.1 , "Child Tasks," for more information.
- KAREL program executes the RUN_TASK built-in. Refer to Subsection 12.4.1 , "Child Tasks," for more information.

In each case, the task will not start running if it requires motion control that is not available.

12.4.1 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.

12.5 TASK CONTROL AND MONITORING

There are three environments from which you can control and monitor tasks:

1. Teach Pendant Programs (TPP) - Subsection 12.5.1
2. KAREL Programs - Subsection 12.5.2
3. KCL commands - Subsection 12.5.3

12.5.1 From TPP Programs

RUN program instruction can be used to run another task. Refer to ARC TOOL OPERATOR's MANUAL for RESUME_PROG instruction.

12.5.2 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 .

- 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.

12.5.3 From KCL

The following KCL commands can be used to control and monitor the status of tasks. Refer to Appendix C , "KCL Command Alphabetic Descriptions," 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.

12.6 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 task, 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 12.6 .
  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 12.6 .
- **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 12.6 .
  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 12.6 .

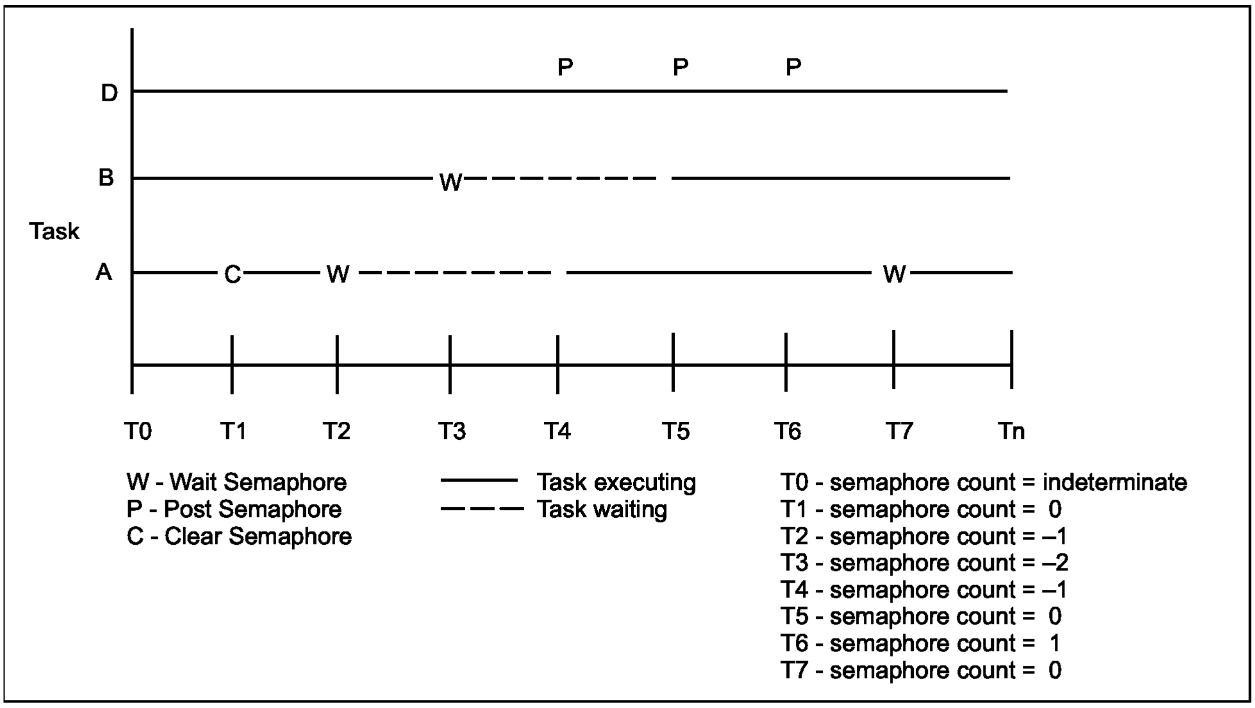


Fig. 12.6 Task synchronization using a semaphore

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:

Example 12.6(a) 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:

Example 12.6(b) 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
```

12.7 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 .

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:

Example 12.7(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:

Example 12.7(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
```

APPENDIX

A KAREL LANGUAGE ALPHABETICAL DESCRIPTION

A.1 OVERVIEW

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.1(a) describes the syntax notation that is used.

Table A.1(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
{ | }	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

Syntax	Meaning	Example	Result
< < | > >	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_typ
```

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.
- **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. Table A.1(b) through Table A.1(h) list the KAREL language elements, described in this appendix, by the type of element. Table A.1(i) lists these elements in alphabetical order.

Table A.1(b) Actions

ABORT Action
Assignment Action
CONTINUE Action
DISABLE CONDITION Action
ENABLE CONDITION Action
NOABORT Action
NOMESSAGE Action
NOPAUSE Action
PAUSE Action
Port_Id Action
PULSE Action
SIGNAL EVENT Action
SIGNAL SEMAPHORE Action
UNPAUSE Action

Table A.1(c) Clauses

EVAL Clause
FROM Clause
IN Clause
WHEN Clause
WITH Clause

Table A.1(d) Conditions

ABORT Condition
CONTINUE Condition
ERROR Condition
EVENT Condition
PAUSE Condition
Port_Id Condition
Relational Condition
SEMAPHORE Condition

Table A.1(e) Data types

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
POSITION Data Type
QUEUE_TYPE Data Type
REAL Data Type
SHORT Data Type
STRING Data Type
STRUCTURE Data Type
VECTOR Data Type
XYZWPR Data Type
XYZWPREXT Data Type

Table A.1(f) Directives

%ALPHABETIZE
%CMOSVARS
%CMOS2SHADOW
%COMMENT
%CRTDEVICE
%DEFGROUP
%DELAY
%ENVIRONMENT
%INCLUDE
%LOCKGROUP
%NOABORT
%NOBUSYLAMP
%NOLOCKGROUP
%NOPAUSE
%NOPAUSESHFT
%PRIORITY
%SHADOWVARS
%STACKSIZE
%TIMESLICE
%TPMOTION
%UNINITVARS

Table A.1(g) KAREL built—in routine summary

Category	Identifier		
Byname	CALL_PROG CALL_PROGLIN	CURR_PROG FILE_LIST	PROG_LIST VAR_INFO VAR_LIST
Data Acquisition	DAQ_CHECKP DAQ_REGPIPE	DAQ_START DAQ_STOP	DAQ_UNREG DAQ_WRITE
Error Code Handling	ERR_DATA	POST_ERR	
File and Device Operation	CHECK_NAME COPY_FILE DELETE_FILE DISMOUNT_DEV FORMAT_DEV	MOUNT_DEV MOVE_FILE PRINT_FILE PURGE_DEV RENAME_FILE	
Serial I/O, File Usage	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
Process I/O Setup	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
KCL Operation	KCL	KCL_NO_WAIT	KCL_STATUS

Category	Identifier		
Memory Operation	CLEAR LOAD LOAD_STATUS	PROG_BACKUP PROG_CLEAR PROG_RESTORE	SAVE SAVE_DRAM
MIRROR	MIRROR		
Program and Motion Control	CNCL_STP_MTN	MOTION_CTL	RESET
Multi-programming	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
Personal Computer Communications	ADD_BYNAMEPC ADD_INTPC ADD_REALPC	ADD_STRINGPC SEND_DATAPC SEND_EVENTPC	
Position	CHECK_EPOS CNV_JPOS_REL CNV_REL_JPOS CURPOS	CURJPOS FRAME IN_RANGE J_IN_RANGE	JOINT2POS POS POS2JOINT SET_PERCH UNPOS
Register Comment Operation	SET_PREG_CMT	SET_REG_CMT	
Queue Manager	APPEND_QUEUE COPY_QUEUE DELETE_QUEUE	GET_QUEUE INIT_QUEUE INSERT_QUEUE	MODIFY_QUEUE
Register Operation	CLR_POS_REG GET_JPOS_REG GET_POS_REG GET_PREG_CMT	GET_REG GET_REG_CMT POS_REG_TYPE SET_EPOS_REG	SET_INT_REG SET_JPOS_REG SET_POS_REG SET_REAL_REG
String Operation	CNV_CONF_STR CNV_INT_STR	CNV_REAL_STR CNV_STR_CONF	CNV_STR_INT CNV_STR_REAL
System	ABS ACOS ARRAY_LEN ASIN ATAN2 BYNAME CHR COS	EXP GET_VAR INDEX INV LN ORD ROUND SET_VAR	SIN SQRT STR_LEN SUB_STR TAN TRUNC UNINIT
Time-of-Day Operation	CNV_STR_TIME CNV_TIME_STR	GET_TIME GET_USEC_SUB	GET_USEC_TIM SET_TIME
TPE Program	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
Translate	TRANSLATE		

Category	Identifier		
User Interface	ACT_SCREEN ADD_DICT ATT_WINDOW_D ATT_WINDOW_S CHECK_DICT CNC_DYN_DISB CNC_DYN_DISE CNC_DYN_DISI CNC_DYN_DISP CNC_DYN_DISR CNC_DYN_DISS DEF_SCREEN	DEF_WINDOW DET_WINDOW DISCTRL_ALPH DISCTRL_FORM DISCTRL_LIST DISCTRL_PLMN DISCTRL_SBMN DISCTRL_TBL FORCE_SPMENU INI_DYN_DISB INI_DYN_DISE INI_DYN_DISI INI_DYN_DISP INI_DYN_DISR INI_DYN_DISS POP_KEY_RD	PUSH_KEY_RD READ_DICT READ_DICT_V READ_KB REMOVE_DICT SET_CURSOR SET_LANG WRITE_DICT WRITE_DICT_V
Vector	APPROACH	ORIENT	

Table A.1(h) Items

CR Input/Output Item

Table A.1(i) Statements

ABORT Statement
Assignment 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
IF..ENDIF Statement
OPEN FILE Statement
OPEN HAND Statement
PAUSE Statement
PROGRAM Statement
PULSE Statement
PURGE CONDITION Statement
READ Statement
RELAX HAND Statement
REPEAT...UNTIL Statement
RETURN Statement
ROUTINE Statement
SELECT..ENDSELECT Statement
SIGNAL EVENT Statement
USING..ENDUSING Statement
WAIT FOR Statement
WHILE..ENDWHILE Statement
WRITE Statement

A.2 - A - KAREL LANGUAGE DESCRIPTION

A.2.1 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: GET_TSK_INFO Built-in. Chapter 6 "CONDITION HANDLER"
Example: Refer to Section B.4 , "Position Data Set and Condition Handlers Program (PTH_MOVE.KL)," for a detailed program example.

A.2.2 ABORT Condition

Purpose: Monitors the aborting of task execution
Syntax : ABORT <PROGRAM[n]>

- 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: CONDITION ... ENDCONDITION Statement, GET_TSK_INFO Built-in, Chapter 6 "CONDITION HANDLER" , Appendix E , “Syntax Diagrams,” for additional syntax information
Example: Refer to the following sections for detailed program examples:
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)

A.2.3 ABORT Statement

Purpose: Terminates task execution.
Syntax : ABORT <PROGRAM[n]>

- After an ABORT, the program cannot be resumed. It must be restarted.

See Also: Appendix E, “Syntax Diagrams,” for additional syntax information
Example: Refer to the following sections for detailed program examples:
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.2.4 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
%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.

NOTE
Do not use more than one motion group in a KAREL program. If you need to use more than one motion group, you must use a teach pendant program.

⚠**WARNING**
Do not run a KAREL program that includes more than one motion group. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: CONT_TASK, RUN_TASK, PAUSE_TASK Built-In Procedures, NO_ABORT Action, %NO_ABORT Translator Directive, Chapter 12 "MULTI TASKING"

Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.2.5 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.

Example: Following example is to execute ABS with INTEGER and REAL. Then it displays the result.

Example A.2.5 ABS built-in function

```
PROGRAM ex_abs
%NOLOCKGROUP

VAR
  int_var     : integer
  real_var    : real
BEGIN
  int_var = abs(-1)
  real_var = abs(-5.4)
  WRITE('abs(-1):', int_var, CR)
  WRITE('abs(-5.4):', real_var,CR)

END ex_abs
```

A.2.6 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.

Example A.2.6 (a) ACOS built-in function

```
routine take_acos
var
  ans_r:  real
begin
  ans_r = acos (-1)
  WRITE ('acos -1 ', ans_r, CR)
END take_acos
```

The second example causes the program to abort since the input value is less than -1 and not within the valid range.

Example A.2.6 (b) ACOS built-in function

```
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
```

A.2.7 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 7.10 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: DEF_SCREEN Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.2.8 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 key, acts on it, and returns it in *action*.
  - ky_cancel (Cancel Table) - All dynamic display is cancelled. 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 cancelled 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 terminattion character was pressed.
  - ky_select - A selectable item was selected.
  - ky_new_menu - A new menu was selected.
  - ky_f1 through ky_f10 - A function key was selected.
- *attach_wind* should be set to TRUE if the table manager needs to be attached to the display device 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 of INIT_TBL.

A.2.9 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: ADD_BYNAMEPC, ADD_INTPC, ADD_REALPC, ADD_STRINGPC
Example: See the following for an example of the ADD_BYNAMEPC built-in.

Example A.2.9 ADD_BYNAMEPC Built-In Procedure

```
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 testbynm
```

A.2.10 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: READ_DICT, WRITE_DICT, REMOVE_DICT Built-In Procedures, Chapter 9 "DICTIONARIES AND FORMS"
Example: Refer to the following sections for detailed program examples:
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.9.1 , "Dictionary Files" (DCALPHEG.UTX)

A.2.11 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: ADD_BYNAMEPC, ADD_INTPC, ADD_REALPC, ADD_STRINGPC
Example: Refer to the TESTDATA example in the built-in function SEND_DATAPC.

A.2.12 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 11 = 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: ADD_BYNAMEPC, ADD_INTPC, ADD_REALPC, ADD_STRINGPC
Example: Refer to the TESTDATA example in the built-in function SEND_DATAPC.

A.2.13 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: ADD_BYNAMEPC, ADD_INTPC, ADD_REALPC, ADD_STRINGPC
Example: Refer to the TESTDATA example in the built-in function SEND_DATAPC.

A.2.14 %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.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.2.15 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: DELETE_QUEUE, INSERT_QUEUE Built-In Procedures. Refer to Section 12.7 , "Using Queues for Task Communication," for more information and an example.

A.2.16 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* .

NOTE
Approach has been left in for older versions of KAREL. You should now directly access the vectors of a POSITION (i.e., posn. approach.)

A.2.17 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 for more information.
- Variable-sized arrays can be defined. Refer to Chapter 2 for more information.

See Also: 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.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.2.18 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.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL), for a detailed program example.

A.2.19 ASIN Built-In Function

Purpose: Returns arcsine (sin-1) 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.

Example A.2.19 (a) ASIN Built-In Function

```
ROUTINE takeasin
VAR
  ans_r:  REAL
BEGIN
  ans_r = ASIN (-1)
  WRITE ('asin -1 ', ans_r, CR)
END takeasin
```

The second example causes the program to abort since the input value is less than -1 and not within the valid range.

Example A.2.19 (b) ASIN Built-In Function

```
ROUTINE takeasin
VAR
  ans_r:  REAL
BEGIN
  ans_r = ASIN (-1.5) -- causes program to abort
  WRITE ('asin -1.5 ', ans_r, CR)
END takeasin
```

A.2.20 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 in Table 11-3, “System Variables Summary”) 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 6 "CONDITION HANDLER", 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.

Example A.2.20 Assignment Action

```
CONDITION[1]:
  WHEN EVENT[1] DO
    DOUT[1] = OFF
    port_var = DOUT[2]
ENDCONDITION
```

A.2.21 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 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 , for detailed information about expressions and their evaluation, Chapter 2 . 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.

Example A.2.21 (a) Assignment Statement

```
int_var = 5
  int_var = 5 + int_var
```

Example: 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.

Example A.2.21 (b) Assignment Statement

```
$SPEED = $SPEED * .25
  array_1[loop_count] = $SPEED
```

Example: 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** .

Example A.2.21 (c) Assignment Statement

```
array_2 = array_1
  struc_var_2 = struc_var_1
```

A.2.22 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.

Example A.2.22 ATAN2 Built-In Function

```
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
END p_atan2                        --returned by ATAN2(100,200)
```

A.2.23 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.2.24 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 7.10 , "User Interface Tips," DET_WINDOW Built-In
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.2.25 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.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.3 - B - KAREL LANGUAGE DESCRIPTION

A.3.1 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.3.1 lists some examples of valid and invalid BOOLEAN values used to represent the Boolean predefined constants.

Table A.3.1 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.1 , "Saving Data to the Default Device" (SAVE_VR.KL)
Section B.3 , "Using Register Built-ins" (REG_EX.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.3.2 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.

A.3.3 BYTE Data Type

Purpose: Defines a variable as a BYTE data type
Syntax : BYTE
Details:
- BYTE has a range of (0 ≤n ≥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.

Example A.3.3 BYTE Data Type

```
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
```

A.3.4 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_byte* 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: Subsection 7.3.1 , “File Attributes”
Example: The following example will clear Port 2 from any bytes still remaining to be read.

Example A.3.4 BYTES_AHEAD Built-In Procedure

```
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
```

A.3.5 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 line feed 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: Subsection 7.10.1 , "User Menu on the Teach Pendant," Subsection 7.10.2 , "User Menu on the CRT/KB"

Example: The following example reads the first number, **oqd_field** , and then uses BYTES_LEFT to determine if the user entered any additional numbers. If so, these numbers are then read.

Example A.3.5 BYTES_LEFT Built-In Function

```
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
```

A.4 - C - KAREL LANGUAGE DESCRIPTION

A.4.1 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: CURR_PROG and CALL_PROGLIN Built-In Functions

A.4.2 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: CURR_PROG and CALL_PROG Built-In Function

Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL), for a detailed program example.

A.4.3 CANCEL Statement

Purpose: Terminates any motion in progress.

Syntax : CANCEL

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.
- 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 cancelled for a different task.

⚠**WARNING**

Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

See Also: Appendix E , ``Syntax Diagrams," for additional syntax information

A.4.4 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: 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.

Example A.4.4 CANCEL FILE Statement

```
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
```

A.4.5 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: ADD_DICT, READ_DICT, WRITE_DICT, REMOVE_DICT Built-In Procedures. Refer to the program example for the DISCTRL_LIST Built-In Procedure and Chapter 9 "DICTIONARIES AND FORMS".
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.4.6 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: GET_POS_FRM

A.4.7 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.4.8 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 ,“ASCII Character Codes”
Example: Refer to the following sections for detailed program examples:
Section B.2 ,"Standard Routines" (ROUT_EX.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.4.9 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.

NOTE
Using carelessly may delete the needed or system programs. Be careful.

Example: The following example clears a KAREL program, clears the variables for a program, and clears a teach pendant program.

Example A.4.9 CLEAR Built-In Procedure
```
-- Clear KAREL program
  CLEAR('test1.pc', status)
  -- Clear KAREL variables
  CLEAR('testvars.vr', status)
  -- Clear Teach Pendant program
  CLEAR('prg1.tp', status)
```

A.4.10 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: POST_SEMA, PEND_SEMA Built-In Procedures, SEMA_COUNT Built-In Function, examples in Chapter 12, "Multi-Tasking"

A.4.11 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: IO_STATUS Built-In Function, Chapter 7 "FILE INPUT/OUTPUT OPERATIONS", , Appendix E , ``Syntax Diagrams," for additional syntax information
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.4.12 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 11 "INPUT/OUTPUT SYSTEM" , for more information on hand signals, Appendix E , ``Syntax Diagrams," for additional syntax information
Example: The following example closes the hand specified by **hand_num** .

Example A.4.12 CLOSE HAND Statement

```
 CLOSE HAND hand_num
```

A.4.13 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: OPEN_TPE Built-In Procedure
Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.4.14 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: I/O-STATUS Built-In Function
Example: Refer to Section B.9, "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.4.15 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 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: GET_PORT_SIM, SET_PORT_SIM Built-In Procedures

A.4.16 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: SET_POS_REG Built-In Procedure, GET_POS_REG Built-In Function
Example: The following example clears the first 100 position registers.

Example A.4.16 CLR_POS_REG Built-In Procedure

```
FOR register_no = 1 to 100 DO
   CLR_POS_REG(register_no, 0, status)
  ENDFOR
```

A.4.17 %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: Subsection A.10.2 IN Clause
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DCLST_EX.KL)

A.4.18 %CMOS2SHADOW Translator Directive

Purpose: Instructs the translator to put all CMOS variables in Shadow memory
Syntax : %CMOS2SHADOW

A.4.19 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 Subsection 7.10.1. "USER Menu on the Teach Pendant" and Subsection 7.10.2 "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: INI_DYN_DISB Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.20 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 CNC_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: INI_DYN_DISE Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.21 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 CNC_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: INI_DYN_DISI Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.22 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 CNC_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: INI_DYN_DISP Built-In Procedure for information on *port_type* codes.
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.23 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 CNC_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: INI_DYN_DISR Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.24 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 CNC_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: INI_DYN_DISS Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.4.25 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: 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.

Example A.4.25 CNV_CONF_STR Built-In Procedure

```
CNV_CONF_STR(posn.pos_config, config_string)
 WRITE('Configuration of posn: ', config_string, cr)
```

A.4.26 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: CNV_STR_INT Built-In Procedure
Example: Refer to the following section for detailed program examples:
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)

A.4.27 CNV_JPOS_REL Built-In Procedure

Purpose: Allows a KAREL program to examine individual joint angles as REAL values
Syntax : CNV_JPOS_REL(jointpos, 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: CNV_REL_JPOS Built-In Procedure
Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.4.28 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: 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.

Example A.4.28 CNV_REAL_STR Built-In Procedure

```
cur_volts = AIN[2]
CNV_REAL_STR(cur_volts, 7, 2, volt_string)
WRITE('Voltage=',volt_string,CR)
```

A.4.29 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.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.4.30 CNV_STR_CONF Built-In Procedure

Purpose: Converts the specified configuration string into a CONFIG data type
Syntax : CNV_STR_CONF(source, targe0t, 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: CNV_CONF_STR Built-In Procedure
Example: The following example sets the configuration of position **posn** to the configuration specified by **config_string**.

Example A.4.30 CNV_STR_CONF Built-In Procedure

```
CNV_STR_CONF(config_string, posn.pos_config, status)
```

A.4.31 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: CNV_INT_STR Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.4.32 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: CNV_REAL_STR Built-In Procedure
Example: The following example converts the STRING **str** into a REAL and puts it into **rate** .

Example A.4.32 CNV_STR_REAL Built-In Procedure

```
REPEAT
 WRITE('Enter rate:')
 READ(str)
 CNV_STR_REAL(str, rate)
UNTIL NOT UNINIT(rate)
```

A.4.33 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: 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** .

Example A.4.33 CNV_STR_TIME Built-In Procedure

```
WRITE('Enter the new time : ')
READ(str_time)
CNV_STR_TIME(str_time,int_time)
SET_TIME(int_time)
```

A.4.34 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:" format.

See Also: GET_TIME Built-In Procedure
Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.4.35 %COMMENT Translator Directive

Purpose: Specifies a comment of up to 16 characters
Syntax : %COMMENT = 'ssssssssssssssss'
where ssssssssssssssss = 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: SET_ATTR_PRG and GET_ATTR_PRG Built-In Procedures
Example: Refer to the following sections for detailed program examples:
Section B.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)

A.4.36 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 Subsection A.24.4 "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 HANDLER", Appendix E , ``Syntax Diagrams," for additional syntax information

Example: Refer to the following sections for detailed program examples:
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)

A.4.37 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.

Example A.4.37. 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.4.38 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_TIM E=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 for additional syntax information, DISCONNECT TIMER Statement

A.4.39 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.

A.4.40 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.

Example A.4.40 CONTINUE Condition

```
CONDITION[1]:
  WHEN CONTINUE DO DOUT[1] = ON
ENDCONDITION
```

A.4.41 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.

  See Also: RUN_TASK, ABORT_TASK, PAUSE_TASK Built-In Procedures, Chapter 12 "MULTI-TASKING".

  Example: The following example prompts the user for the task name and continues the task execution. Refer to Chapter 12 , for more examples.

Example A.4.41 CONT_TASK Built-In Procedure

```
PROGRAM conttsk
%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 conttsk
```

A.4.42 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: RENAME_FILE, DELETE_FILE Built-In Procedures
Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.4.43 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: APPEND_QUEUE, DELETE_QUEUE, INSERT_QUEUE Built-In Procedures, Section 12.7 , "Using Queues for Task Communication"
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.

Example A.4.43 COPY_QUEUE Built-In Procedure

```
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
```

A.4.44 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: CREATE_TPE Built-in Procedure
Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.4.45 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 ±18000 degrees. Otherwise, the program will be aborted with an error.

Example: Refer to TAN Built-in Procedure.

A.4.46 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.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.2 ,"Standard Routines" (ROUT_EX.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.4.47 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: SET_ATTR_PRG Built-In Procedure

A.4.48 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.

⚠**CAUTION**
It makes debugging difficult to create variable that doesn't exist in source code.
Do not use CREATE_VAR Built-in Procedure except when absolutely necessary.

See Also: CLEAR, RENAME_VAR Built-In Procedures

A.4.49 %CRTDEVICE

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.

A.4.50 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 over travel
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 over travel.

NOTE
axis_limit_mask and *ovr_trv_mask* are not available in this release and can be set to 0.

⚠**WARNING**
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

See Also: CURPOS Built-In Function
Example: The following example gets the current joint position of the robot.

Example A.4.50 CURJPOS Built-In Function

```
PROGRAM getpos
VAR
 jnt: JOINTPOS
BEGIN
 jnt=CURJPOS(0,0)
END getpos
```

A.4.51 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 over travel
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 over travel.

NOTE
axis_limit_mask and *ovr_trv_mask* are not available in this release and will be ignored if set.

WARNING
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL)

A.4.52 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 ≥12

A.5 - D - KAREL LANGUAGE DESCRIPTION

A.5.1 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, i.e., 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: DAQ_WRITE Built-In.
Example: Refer to the DAQWRITE example in the built-in function DAQ_WRITE.

NOTE
This built-in is only available when data monitor options are loaded.

A.5.2 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: DAQ_UNREG Built-In.
Example: The following example registers KAREL pipe 1 to write a variable in the program.

Example A.5.2 DAQ_REGPIPE Built-In Procedure

```
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
```

NOTE
This built-in is only available when data monitor options are loaded.

A.5.3 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.

See Also: DAQ_REGPIPE Built-In and DAQ_STOP Built-In,
Example: The following example starts KAREL pipe 1 in streaming mode.

Example A.5.3 DAQ_START Built-In Procedure

```
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
```

NOTE
This built-in is only available when data monitor options are loaded.

A.5.4 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: DAQ_START Built-In.
Example: Refer to the PIPONOFF example in the built-in function DAQ_START.

NOTE
This built-in is only available when data monitor options are loaded.

A.5.5 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 un-register the pipe before re-registering using DAQ_REGPIPE.

See Also: DAQ_REGPIPE Built-In.
Example: The following example unregisters KAREL pipe 1.

Example A.5.5 DAQ_UNREG Built-In Procedure

```
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
```

NOTE
This built-in is only available when data monitor options are loaded.

A.5.6 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: DAQ_REGPIPE and DAQ_CHECKP.
Example: The following example registers KAREL pipe 2 and writes to it when the pipe is active.

Example A.5.6 DAQ_WRITE Built-In Procedure

```
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
```

NOTE
This built-in is only available when data monitor options are loaded..

A.5.7 %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.

⚠WARNING
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

A.5.8 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: ACT_SCREEN Built-In Procedure
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.5.9 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: ATT _WINDOW_D, ATT_WINDOW_S Built-In Procedures

A.5.10 %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.5.11 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.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.5.12 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: COPY_FILE, RENAME_FILE Built-In Procedures
Example: Refer to Section B.1 ,"Saving Data to the Default Device" (SAVE_VRS.KL), for a detailed program example.

A.5.13 DELETE_QUEUE Built-In Procedure

Purpose: Deletes an entry from a queue
Syntax : DELETE_QUEUE(sequence_no, queue, 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: APPEND_QUEUE, COPY_QUEUE, INSERT_QUEUE Built-In Procedures, Section 12.7 , "Using Queues for Task Communication"

A.5.14 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: CREATE_TPE, CLOSE_TPE, COPY_TPE, OPEN_TPE, SELECT_TPE Built-In Procedures

A.5.15 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: DEF_WINDOW, ATT_WINDOW_S, ATT_WINDOW_D Built-In Procedures
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.5.16 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.

Example A.5.16 DISABLE CONDITION Action

```
CONDITION[1]:
  WHEN EVENT[1] DO
  DISABLE CONDITION[2]
ENDCONDITION
```

A.5.17 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 HANDLER" , 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** .

Example A.5.17 DISABLE CONDITION Statement

```
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]
  ENDCONDITION
  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
```

A.5.18 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 sets the INTEGER variable **timevar** to 0 and connects the timer. After DO[1] is ON, the timer is disconnected.

Example A.5.18 DISCONNECT TIMER Statement

```
PROGRAM EX_DISC
%NOLOCKGROUP
VAR
  timevar:INTEGER
BEGIN
  timevar = 0
  CONNECT TIMER TO timevar
  WAIT FOR DOUT[1] = ON
  DISCONNECT TIMER timevar
  WRITE('timevar:', timevar, CR)
END EX_DISC
```

A.5.19 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 Subsection 7.10.1 , "User Menu on the Teach Pendant," Subsection 7.10.2 , "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 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: ACT_SCREEN, DISCTRL_LIST Built-In Procedures
Example: Refer to Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCLAP_EX.KL), for a detailed program example.

A.5.20 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, inact_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 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 9 "DICTIONARIES AND FORMS" , for more details and examples.

A.5.21 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: DISCTRL_FORM Built-In Procedure, Subsection 7.10.1 , "User Menu on the Teach Pendant," Subsection 7.10.2 , "User Menu on the CRT/KB," Chapter 9 "DICTIONARIES AND FORMS"

Example: Refer to Section B.9 , "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.5.22 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 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.

Example A.5.22 DISCTRL_PLMN Built-In Procedure

```
----------------------------------------------
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
```

A.5.23 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 'subm' 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 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.

Example A.5.23 DISCTRL_SBMN Built-In Procedure

```
--------------------------------------------
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
```

A.5.24 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 INIT_TBL Built-In routine 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 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 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)
- *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 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 INIT_TBL Built-In routine for an example of setting up the dictionary text and initializing the parameters.

A.5.25 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: MOUNT_DEV, FORMAT_DEV Built-In Procedures

Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.5.26 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.9 , "Displaying a List From a Dictionary File" (DCLIST_EX.KL), for a detailed program example.

A.6 - E - KAREL LANGUAGE DESCRIPTION

A.6.1 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: DISABLE CONDITION Action, Chapter 6 "CONDITION HANDLER"

A.6.2 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: DISABLE CONDITION Statement, Chapter 6 "CONDITION HANDLER" , Appendix E , ``Syntax Diagrams," for additional syntax information
Example: Refer to the following sections for detailed program examples.
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOV.KL)

A.6.3 %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.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)

A.6.4 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.g., E-STOP), *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: POST_ERR Built-In Procedure

A.6.5 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 HANDLER" , for more information on using conditions. The appropriate application-specific Operator's Manual for a list of all error codes

A.6.6 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 HANDLER" „ for more information on using conditions

A.6.7 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: SIGNAL EVENT Action, and CONDITION or SIGNAL EVENT Statement

A.6.8 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 + timevar/(timevar + 20))** .

Example A.6.8 EXP Built-In Function

```
PROGRAM ex_exp
%NOLOCKGROUP
VAR
  timevar     : real
   distance : real
BEGIN
  WRITE (CR, 'Enter time needed for move:')
  READ (timevar)
  distance = timevar *
             EXP(-6.44 + timevar/(timevar + 20))
  WRITE (CR, CR, 'Distance for move:', distance)
END ex_exp
```

A.7 - F - KAREL LANGUAGE DESCRIPTION

A.7.1 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 a static variable.
  - 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.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.7.2 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 only returned 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: VAR_LIST, PROG_LIST Built-In Procedures
Example: Refer to Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL), for a detailed program example.

A.7.3 FOR...ENDFOR Statement

Purpose: Looping construct based on an INTEGER counter
Syntax : FOR count = initial || TO | DOWNTO || final
DO{stmnt} ENDFOR
where:
[in]count :INTEGER variable
[in]initial :INTEGER expression
[in]final :INTEGER expression
[in]stmnt :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 DOWNTO is used, *count* is decremented for each loop iteration.
- If *count* is greater than *final* using TO, *stmnt* is never executed.
- If *count* is less than *final* using DOWNTO, *stmnt* 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.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.7.4 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_TPALARM, SCR_ALM_ALL — ALARM Alarm Log
  SPI_TPALARM, SCR_ALM_MOT — ALARM Motion Log
  SPI_TPALARM, SCR_ALM_SYS — ALARM System Log
  SPI_TPALARM, 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_TPMMACRO, SCR_MACSETUP — SETUP Macro
  SPI_TPREFPOS — SETUP Ref Position
  SPI_TPPWORD — SETUP Passwords
  SPI_TPHCCOMM — 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_TPPOSN — 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

> **NOTE**
> Use the value described above only.

See Also: ACT_SCREEN Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.2 ,"Standard Routines" (ROUT_EX.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.12 , "Dictionary Files" (DCLISTEG.UTX)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)

A.7.5 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: MOUNT_DEV, DISMOUNT_DEV Built-In Procedures
Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.7.6 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.7.6 .
- 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.7.6 .
- 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.

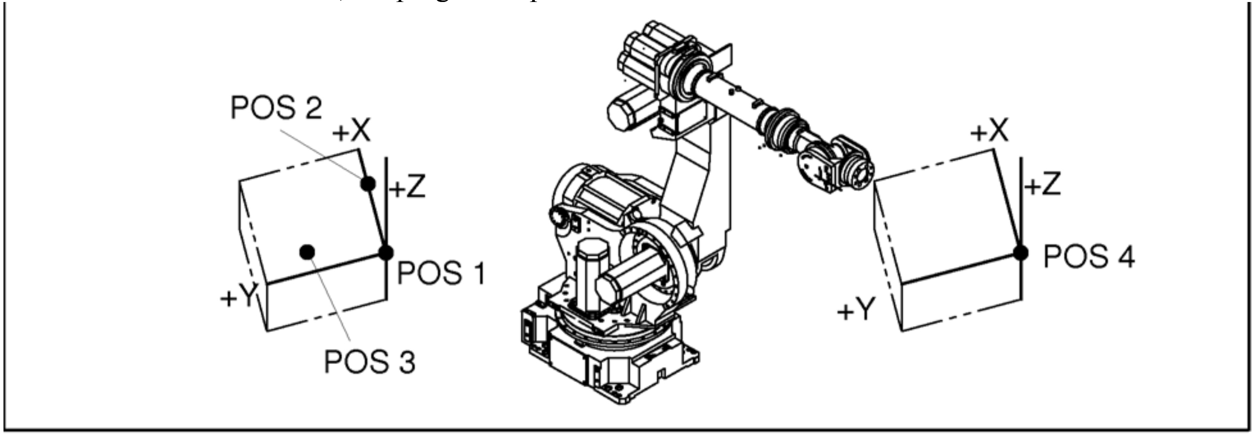


Fig. A.7.6 FRAME Built-In Function

A.7.7 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.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.8 - G - KAREL LANGUAGE DESCRIPTION

A.8.1 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.13.4 .
  AT_PROTECT : Protection code; 1 :Protection OFF ; 2 :Protection ON
  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: SET_ATTR_PRG, GET_TSK_INFO, SET_TSK_ATTR Built-In Procedures

A.8.2 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.

A.8.3 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: GET_POS_REG, SET_JPOS_REG, SET_POS_REG Built-in Procedures
Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL) for a detailed program example.

A.8.4 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: SET_JPOS_TPE, GET_POS_TPE, SET_POS_TPE Built-ins

Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.8.5 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 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.

Example A.8.5 GET_PORT_ASG Built-In Procedure

```
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
```

A.8.6 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_sbits :Stop bit
  atr_dbits :Data length
  atr_xonoff :Xon/Xoff
  atr_eol :End of 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: 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.

Example A.8.6 GET_PORT_ATR Built-In Function

```
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_sbits,atr_value)
 IF(atr_value <> SBITS_1) THEN
  stat=SET_PORT_ATR(port_2,atr_sbits,SBITS_1)
 ENDIF
-- sets the data bit to 5, if not already set
 stat=GET_PORT_ATR(port_2,atr_dbits,atr_value)
 IF(atr_value <> DBITS_5) THEN
  stat=SET_PORT_ATR(port_2,atr_dbits,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_atr
```

A.8.7 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 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: GET_PORT_VAL, GET_PORT_MOD, SET_PORT_CMT, SET_PORT_VAL, SET_PORT_MOD Built-in Procedures.

A.8.8 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.

Example A.8.8 GET_PORT_MOD_Built-In Procedure

```
PROGRAM getmodprog
%ENVIRONMENT IOSETUP
%INCLUDE FR:\kliotyps
 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
```

A.8.9 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 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: GET_PORT_MOD, SET_PORT_SIM, SET_PORT_MOD Built-in Procedures.

A.8.10 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 value is being returned. Codes are defined in KLIOTYPS.KL.
- *port_no* specifies the port number whose value is being set.
- *value* is returned with the current value (status) of the specified port. For BOOLEAN port types, (i.e. DIN), this will be 0 = OFF, or 1 = ON.
- *status* is returned with zero if the parameters are valid and the value is returned for the specified port.

See Also: GET_PORT_CMT, GET_PORT_MOD, SET_PORT_CMT, SET_PORT_VAL, SET_PORT_MOD Built-in Procedures.

A.8.11 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: GET_POS_TYP, CHECK_EPOS.

A.8.12 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: GET_JPOS_REG, SET_JPOS_REG, SET_POS_REG, GET_REG Built-in Procedures.
Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL) , for a detailed program example.

A.8.13 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: GET_JPOS_TPE, SET_JPOS_TPE, SET_POS_TPE, GET_POS_TYP Built-in Procedures.
Example: Refer to Section B.11, "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.8.14 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.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_T.KL), for a detailed program example.

A.8.15 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
%ENVIORNMENT 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.8.16 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: MODIFY_QUEUE Built-In Procedure, Section 12.7, "Using Queues for Task Communication"
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.

Example A.8.16 GET_QUEUE Built-In Procedure

```
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
```

A.8.17 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.3 ,"Using Register Built-ins" (REG_EX.KL) , for a detailed program example.

A.8.18 GET_REG_CMT

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.8.19 GET_SREG_CMT Built-In Procedure

Purpose: To retrieve the comment information based on a given string register number.

> **NOTE**
> String register is available only on 7DA5 series or later.

Syntax : GET_SREG_CMT(register_no, comment, status)
Input/Output Parameters:
[in] register_no : INTEGER
[out] comment : STRING
[out] status :INTEGER
%ENVIRONMENT Group :REGOPE
Details:

- *Register_no* specifies which string register to retrieve the comment from.
- The comment of the given string register is returned in *comment*.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: GET_STR_REG, SET_STR_REG, SET_SREG_CMT Built-In Procedure

A.8.20 GET_STR_REG Built-In Procedure

Purpose: To retrieve the string data based on a given string register number.

> **NOTE**
> String register is available only on 7DA5 series or later.

Syntax : GET_STR_REG(register_no, value, status)
Input/Output Parameters:
[in] register_no : INTEGER
[out] value : STRING
[out] status :INTEGER
%ENVIRONMENT Group :REGOPE
Details:

- *Register_no* specifies which string register to retrieve the value from.

- The value of the given string register is returned in *value*.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: GET_STR_REG, SET_STR_REG, SET_SREG_CMT Built-In Procedure

A.8.21 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.8.21 INTEGER Representation of Current Time

Bit	Comment
31-25	year
24-21	month
20-16	day
15-11	hour
10-5	minute
4-0	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: CNV_TIME_STR Built-In Procedure
Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.8.22 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: SET_TPE_CMT and OPEN_TPE for more Built-in Procedures.

A.8.23 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: Application-specific Operator's Manual, 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.

Example A.8.23. GET_TPE_PRM Built-In Procedure

```
    Example calling from TPE program
    1:   CALL WRTPPRM('AGM',5,15.7) ;

----------------------
PROGRAM WRTPPRM
----------------------
%NOLOCKGROUP

VAR
    dummy_int : integer
    dummy_real : real
    dummy_str : string[10]
    status : INTEGER
    data_type : integer
    int_value : integer
    real_value : REAL
    str_value : string[30]
BEGIN
    -- Reat the 1st parameter of TP program CALL.
    GET_TPE_PRM(1, data_type, dummy_int, dummy_real, str_value, status)
    if status <> 0 then
        write ('GET PARM(1) failed:', status, CR)
        post_err(status, '',0,0)
    else
        if data_type <> 3 then -- The first param is dictionary name. It must be
string.
            write ('data type of param1 is NOT string:', data_type, CR)
        else
            -- Read the 2nd parameter of TP program CALL.
            GET_TPE_PRM(2, data_type, int_value, dummy_real, dummy_str,
status)
            if status <> 0 then
                write ('GET PARM(2) failed:', status, CR)
                post_err(status,'',0,0)
            else
                if data_type <> 1 then --INTEGER?
                    write ('data type of param2 is NOT integer:', data_type, CR)
                    post_err(status,'',0,0)
                else
                    -- Read the 3rd parameter.
                    GET_TPE_PRM(3, data_type, dummy_int, real_value,
dummy_str, status)
                    if data_type <> 2 then -- real?
                      write('data type of param2 is NOT real:', data_type, CR)
                      post_err(status,'',0,0)
                    else
                     write ('1st parameter:', str_value,CR)
                     write ('2nd parameter:', int_value,CR)
                     write ('3rd parameter:', real_value,CR)
                    endif
                  endif
                endif
              endif
            endif
END WRTPPRM
```

A.8.24 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 12 "MULTI-TASKING"
Example: See examples in Chapter 12 "MULTI-TASKING"

A.8.25 GET_USEC_SUB Built-In Procedure

Purpose: Returns an INTEGER value indicating the elapsed time in microseconds.
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)
```

Fig. A.8.25 GET_USEC_SUB Built-In Function

A.8.26 GET_USEC_TIM Built-In Function

Purpose: Returns an INTEGER value indicating the current time in microseconds from within the KAREL system.
Syntax: us = GET_USEC_TIM
Function Return Type: INTEGER
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.8.27 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 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: SET_VAR Built-In Procedure, "FANUC Robot series R-30*i*A/R-30*i*A Mate CONTROLLER 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 Example A.8.27(a) .

Example A.8.27 (a) Accessing $TP_DEFPROG on MHROB03

```
GET_VAR(entry, '¥¥MHROB03¥*system*', '$TP_DEFPROG', strvar, status)
```

Example 2: Example A.8.27(B) displays two programs, **getvar01** and **getvar02** . The program **getvar01** uses a FOR loop to increment the value of the INTEGER variable **num_of_parts** . **getvar01** also assigns values to the ARRAY **part_array** . The program **getvar02** 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** .

Example A.8.27 (b) GET_VAR Built-In Procedure

```
PROGRAM getvar01
  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 getvar01


PROGRAM getvar02
  VAR
    entry, status    : INTEGER
    count, new_count : INTEGER
    part_name       : STRING[20]
  BEGIN
    GET_VAR(entry, 'getvar01', 'part_array[3]', part_name, status)
    WRITE('Part Name is Now....>', part_name, cr)

    GET_VAR(entry, 'getvar01', 'num_of_parts', count, status)
    WRITE('COUNT Now Equals....>', count, cr)

    GET_VAR(entry, 'getvar02', 'count', new_count, status)

  END getvar02
```

A.8.28 GO TO Statement

Purpose: Transfers control to a specified statement
Syntax : || GO TO | GOTO || stmnt_label
where:
stmnt_label : A valid KAREL identifier
Details:

- *stmnt_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: Subsection 2.1.5 , ``Labels,'' for more information on rules for labels, Appendix E , ``Syntax Diagrams,'' for additional syntax information
Example: The following example switches messages according to DI[1] and DI[2].

Example A.8.28 GO TO Statement

```
PROGRAM ex_goto
%NOLOCKGROUP
BEGIN
  IF NOT DIN[1] THEN
    GO TO end_it
  ENDIF
  IF NOT DIN[2] THEN
    GO TO end_it
  ENDIF
  WRITE('Both DIN[1] and DIN[2] are ON',CR)
  ABORT
END_IT::
  WRITE('Either DIN[1] or DIN[2] is not ON',CR)
END ex_goto
```

A.9 - H - KAREL LANGUAGE DESCRIPTION

There are no KAREL descriptions beginning with "H".

A.10 - I - KAREL LANGUAGE DESCRIPTION

A.10.1 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.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)

Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.10.2 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 %SHADOW 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: Subsection 1.4.1, %CMOSVARS Translator Directive, %SHADOW Translator Directive, %SHADOWCMOS Translator Directive
Example: Refer to the following sections for detailed program examples:
In DRAM,
In CMOS, Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL) or Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)

A.10.3 %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 12 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 klevkmsk
  %INCLUDE klevkeys
  %INCLUDE klevccdf
  %INCLUDE 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 copied to the hard disk as part of the installation process.

Example: Refer to the following sections for detailed program examples:

Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)

A.10.4 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** .

Example A.10.4 INDEX Built-In Function

```
class_key = 'Old'
  part_desc = 'Refurbished Old Part'
  IF INDEX(part_desc, class_key) > 0 THEN
    in_class = TRUE
  ENDIF
```

A.10.5 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 Subsection 7.10.1 , "User Menu on the Teach Pendant," Subsection 7.10.2 , "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
  8 :Reverse video
- To have multiple display attributes, use the OR operator to combine the constant attribute values together.
- *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 then 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: CNC_DYN_DISB built-in procedure
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.6 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 *string s* must be static (not local) variables.
- Refer to the INI_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: CNC_DYN_DISE built-in procedure
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.7 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.10.7 .

Table A.10.7 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.dddddd, 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.dddddde+-xx or [-]m.ddddddE+-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 INI_DYN_DISB built-in procedure for a description of the other parameters listed above.

See Also: CNC_DYN_DISI, DEF_WINDOW Built-In Procedure
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.8 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 (e.g., 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 (e.g., 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_DISB built-in procedure for a description of other parameters listed above.

See Also: CNC_DYN_DISP Built-In procedure
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.9 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: CNC_DYN_DISR Built-In Procedure
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.10 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: CNC_DYN_DISS, INI_DYN_DISI Built-In Procedures
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.10.11 INIT_QUEUE Built-In Procedure

Purpose: Sets a queue variable entry to have no entries in the queue
Syntax : INIT_QUEUE(queue)
Input/Output Parameters:
[out] queue_t :QUEUE_TYPE
%ENVIRONMENT Group : PBQMGR
Details:

- queue_t is the queue to be initialized

See Also: GET_QUEUE, MODIFY_QUEUE Built-In Procedures, QUEUE_TYPE Data Type, Section 12.7 , "Using Queues for Task Communication"
Example: The following example initializes a queue called **job_queue.**

Example A.10.11 INIT_QUEUE Built-In Procedure

```
PROGRAM init_queue
%environment PBQMGR
VAR
  job_queue FROM globals: QUEUE_TYPE
BEGIN
  INIT_QUEUE(job_queue)
END init_queue
```

A.10.12 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:

- 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.
- 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.
- *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 2 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:

```
$ 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
" >"
```

- 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:

```
$ example_sub_win_enum_c
"Red"
$
"Blue"
$
"Green"
$
"Yellow"
$
"\a"
```

- 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:

```
$ example_boolean_c
"OFF" &new_line
"ON"
```

- *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.10.12 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.dddddd, 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.dddddde+-xx or [-]m.ddddddE+-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 %.

Example:

```
"%d"
"%-10s"
```

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 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 2 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_KST_ITEM: ARRAY OF STRING containing the desired values.
    - TPX_KSL_ITEM: ARRAY OF STRING containing the desired values. These values cannot be edited by the user. If one is selected, ACT_TBL will return.
    - 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. A.10.12.(a) is the example that is used by A.10.12(b).

Example A.10.12 (a) INIT_TBL Built-In Procedure

```
----------------------------------------------
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"
$
"......"
```

Example A.10.12 (b) INIT_TBL Built-In Procedure

```
---------------------------------------------
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_updt, 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
```

A.10.13 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: CHECK_EPROS 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.

Example A.10.13. IN_RANGE Built-In Function

```
IF IN_RANGE(pallet :  part_slot) THEN
    WRITE('I can get there!',CR)
   ELSE
    WRITE('I can't get there!',CR)
   ENDIF
```

A.10.14 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, 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: MODIFY_QUEUE, APPEND_QUEUE, DELETE_QUEUE Built-In Procedures, Section 12.7 , "Using Queues for Task Communication"
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.

Example A.10.14 INSERT_QUEUE Built-In Procedure

```
PROGRAM ins_queue
%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
```

A.10.15 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 +2147483646.
- 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.10.15)

Table A.10.15 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.10.16 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.

Example A.10.16 INV Built-In Function

```
PROGRAM inv01
%NOLOCKGROUP
VAR
  rack_pos, part_pos : POSITION
  p1 : POSITION
BEGIN
  -- rack_pos and part_pos should be initialized
  p1 = INV(rack_pos):part_pos
END inv01
```

A.10.17 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 as follows:

Example: The following example returns to the caller the module in the specified rack and slot number.

Example A.10.17 IO_MOD_TYPE Built-In Procedure

```
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
```

A.10.18 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. Some of the common errors are shown in Table A.10.18 .

Table A.10.18 IO_STATUS Errors

Error Code	Comment
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, the program is aborted with an error.

Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLIST_EX.KL), for a detailed program example.

A.11 - J - KAREL LANGUAGE DESCRIPTION

A.11.1 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: IN_RANGE Built-in Function, CHECK_EPROS Built-in procedure

A.11.2 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 4 + 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.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.11.3 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.12 - K - KAREL LANGUAGE DESCRIPTION

A.12.1 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: KCL_NO_WAIT, KCL_STATUS Built-In Procedures
Example: The following example will show programs and wait until finished. Status will be the outcome of this operation.

Example A.12.1 KCL Built-In Procedure

```
PROGRAM kcl_test
VAR
 command :STRING[20]
 status :INTEGER
BEGIN
 command = 'SHOW PROGRAMS'
 KCL (command, status)
END kcl_test
```

Example: Refer to Example 9.3.18(f) for another example.

A.12.2 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: KCL, KCL_STATUS Built-In Procedures
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.

Example A.12.2 KCL_NO_WAIT Built-In Procedure

```
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
```

A.12.3 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: KCL_NO_WAIT, KCL Built-In Procedures

A.13 - L - KAREL LANGUAGE DESCRIPTION

A.13.1 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** .

Example A.13.1 LN Built-In Function

```
PROGRAM ln01
%NOLOCKGROUP
%NOPAUSESHFT
VAR
  real_var   : real
   ln_value : real
BEGIN
  WRITE(CR, CR, 'enter a number =')
  READ(real_var)
  ln_value = LN(real_var)
  WRITE('LN(',real_var,') is ', ln_value, CR)
END ln01
```

A.13.2 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.

Example: Refer to the following sections for detailed program examples:
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)

A.13.3 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.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)

A.13.4 %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.

⚠**WARNING**
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

See Also: %NOLOCKGROUP Directive

A.14 - M - KAREL LANGUAGE DESCRIPTION

A.14.1 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: The appropriate application-specific Operator's Manual , Section on "Mirror Shift Function"

A.14.2 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 set in 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: COPY_QUEUE, GET_QUEUE, DELETE_QUEUE Built-In Procedures Section 12.7 , "Using Queues for Task Communication"
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**).

Example A.14.2 MODIFY_QUEUE Built-In Procedure

```
PROGRAM mod_queue
%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
```

A.14.3 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.14.3 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.14.3 , 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".

⚠**WARNING**
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

- Returns TRUE if the KAREL program has motion control for the specified group of axes.

A.14.4 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: DISMOUNT_DEV and FORMAT_DEV
Example: Refer to Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL), for a detailed program example.

A.14.5 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 FROM disk.

Example A.14.5 MOVE_FILE Built-In Procedure

```
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
```

A.14.6 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 "FANUC Robot series R-30*i*A/R-30*i*A Mate CONTROLLER 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.

Example A.14.6 MSG_CONNECT Built-In Procedure

```
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 tcpserv8
```

A.14.7 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 "FANUC Robot series R-30*i*A/R-30*i*A Mate CONTROLLER Ethernet Function OPERATOR'S MANUAL(B-82974EN)".
Example: Refer to MSG_CONNECT Built-In Procedure for more examples.

A.14.8 MSG_PING Built-In Procedure

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 "FANUC Robot series R-30*i*A/R-30*i*A Mate CONTROLLER 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.

Example A.14.8 MSG_PING Built-In Procedure

```
PROGRAM pingtest
VAR
 Tmp_int  : INTEGER
 Status   : integer
BEGIN
  WRITE('pinging..',cr)
  MSG_PING('fido',status)
  WRITE('ping Status = ',status,cr)
END pingtest
```

A.15 - N - KAREL LANGUAGE DESCRIPTION

A.15.1 NOABORT Action

Purpose: Prevents program execution from aborting when an external error occurs

Details:

- The NOABORT action usually corresponds to an ERROR[n].
- If the program is aborted by itself (i.e., 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.

Example A.15.1 NOABORT Action

```
PROGRAM noabort
%NOLOCKGROUP
BEGIN
  --Pulse Mismatch condition handler
  CONDITION[801]:
    WHEN ERROR[11038] DO
    NOABORT
  ENDCONDITION
  ENABLE CONDITION[801]
END noabort
```

A.15.2 %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.15.3 %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.15.4 %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 cancelled.
- 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.

⚠**CAUTION**
Many of the fields in the $GROUP system variable are initialized to a set of default values when a KAREL task is started. When %NOLOCKGROUP is specified, this initialization does not occur. Therefore, when a motion is issued the current values of these fields are used. This could cause unexpected motion.

⚠**WARNING**
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

See Also: %LOCKGROUP Translator Directive
Example: Refer to the following sections for detailed program examples:
Section B.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)

A.15.5 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.

A.15.6 NOPAUSE Action

Purpose: Prevents program execution from pausing
Syntax : NOPAUSE
Details:

- The NOPAUSE action usually corresponds to an ERROR[n] or PAUSE condition.
- 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.

Example A.15.6 NOPAUSE Action

```
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
```

A.15.7 %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.15.8 %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.16 - O - KAREL LANGUAGE DESCRIPTION

A.16.1 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_ATR 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: IO_STATUS Built-In Function, SET_FILE_ATR Built-In Procedure, Chapter 7 "FILE INPUT/OUTPUT OPERATIONS", Chapter 9 "DICTIONARIES AND FORMS", Appendix E , ``Syntax Diagrams" for more syntax information
Example: Refer to the following sections for detailed program examples:
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)

A.16.2 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 11, ``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 opens the hand of the robot specified by the INTEGER variable **hand_num** .

Example A.16.2 OPEN HAND Statement

```
OPEN HAND hand_num
```

A.16.3 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: CREATE_TPE Built-In Procedure, COPY_TPE Built-In Procedure, AVL_POS_NUM Built-In Procedure
Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.16.4 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.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.16.5 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.17 - P - KAREL LANGUAGE DESCRIPTION

A.17.1 PAUSE Action

Purpose: Suspends execution of a running task
Syntax : PAUSE <PROGRAM[n]>
Details:

- The PAUSE action pauses task execution in the following manner:
    - 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: UNPAUSE Action

A.17.2 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 enables the condition handler.

Example A.17.2. PAUSE Condition

```
CONDITION[1]:
  WHEN PAUSE DO
    DOUT[1] = TRUE
    CONTINUE
    ENABLE CONDITION[1]
ENDCONDITION
```

A.17.3 PAUSE Statement

Purpose: Suspends execution of a KAREL program
Syntax : PAUSE < PROGRAM [n] >
Details:

- The PAUSE statement pauses program execution in the following manner:
  - 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.

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.

Example A.17.3. PAUSE Statement

```
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
```

A.17.4 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: RUN_TASK, CONT_TASK, ABORT_TASK Built-In Procedures, Chapter 12 "MULTI-TASKING"

Example: The following example pauses the user-specified task and stops any motion. Refer to Chapter 12 "MULTI-TASKING" , for more examples.

Example A.17.4 PAUSE_TASK Built-In Procedure

```
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
```

A.17.5 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: POST_SEMA, CLEAR_SEMA Built-In Procedures, SEMA_COUNT Built-In Function, Chapter 12 "MULTI-TASKING"

Example: See examples in Chapter 12 "MULTI-TASKING"

A.17.6 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 for internal use. Set form_dict to ''.
- form_ele is for internal use. Set form_ele to 0.
- status explains the status of the attempted operation. If it is not equal to 0, then an error occurred.

See Also: Chapter 8 "FILE SYSTEM"

A.17.7 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: PUSH_KEY_RD, READ_KB Built-in Procedures
Example: Refer to the example for the READ_KB Built-In Routine.

A.17.8 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 HANDLER" , Chapter 7 "FILE INPUT/OUTPUT OPERATIONS", , Relational Conditions, Appendix A, ``KAREL Language Alphabetical Description"

A.17.9 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 HANDLER" , Chapter 7 "FILE INPUT/OUTPUT OPERATIONS", , Relational Conditions, Appendix A, ``KAREL Language Alphabetical Description"

A.17.10 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±10000000 mm (±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 ±180 degrees. Otherwise, the program is paused with an error.

A.17.11 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, and 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.17.12 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 : 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: GET_POS_REG, GET_JPOS_REG, SET_POS_REG, SET_JPOS_REG Built-in Procedures
Example: The following example determines the position type in the register and uses the appropriate built-in to get data.

Example A.17.12 POS_REG_TYPE Built-In Procedure

```
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
```

A.17.13 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 56 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:

Example A.17.13 POSITION Data Type

```
POSITION = STRUCTURE
    NORMAL:  VECTOR         -- read-only
    ORIENT:  VECTOR         -- read-only
    APPROACH:  VECTOR        -- read-only
    LOCATION:  VECTOR        -- read-write
    CONFIG_DATA:  CONFIG      -- read-write
  ENDSTRUCTURE
```

See Also: POS, UNPOS Built-In Functions

A.17.14 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 all motion
  2 : ABORT, all tasks and cancel

See Also: ERR_DATA Built-In Procedure, the appropriate application-specific Operator's Manual, "Alarm Code List".
Example: Refer to Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL) , for a detailed program example.

A.17.15 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: PEND_SEMA, CLEAR_SEMA Built-In Procedures, SEMA_COUNT Built-In Function, Chapter 12 "MULTI-TASKING" ,
Example: See examples in Chapter 12 "MULTI-TASKING"

A.17.16 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.17.17 %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 12.3 , "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.

A.17.18 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_JBTYPE :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, " FILE-002 Device is Full". In this case a partial backup 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 builtins 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)
```

Example : 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)
```

Example : 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.17.19 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_JBTYPE :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)
- *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 use must use SELECT_TPE() built-in to select a program which is not in the clear set.
- The PROG_BACKUP, PROG_CLEAR and PROG_RESTORE builtins 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)
```

Example: 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.17.20 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 builtins 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)
```

Example: 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)
```

Example: 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.17.21 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* is used 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 characters
- *prog_name* = 12 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: VAR_LIST Built-In Procedure
Example: Refer to the following sections for detailed program examples:
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.17.22 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.17.23 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 HANDLER" , Chapter 7 "FILE INPUT/OUTPUT OPERATIONS",

A.17.24 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.

Example A.17.24 PULSE Statement

```
PULSE DOUT[5] FOR (seconds * 1000) NOWAIT
PULSE DOUT[start_air] FOR 50 NOWAIT
```

A.17.25 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 HANDLER" , 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.

Example A.17.25 PURGE CONDITION Statement

```
IF ignore_cond THEN
  PURGE CONDITION[1]
ELSE
  ENABLE CONDITION[1]
ENDIF
```

A.17.26 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: POP_KEY_RD Built-In Procedure
Example: Refer to the READ_KB Built-In Procedure for an example.

A.18 - Q - KAREL LANGUAGE DESCRIPTION

A.18.1 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: APPEND_QUEUE, DELETE_QUEUE, INSERT_QUEUE, COPY_QUEUE, GET_QUEUE, INIT_QUEUE, MODIFY_QUEUE Built-In Procedures

A.19 - R - KAREL LANGUAGE DESCRIPTION

A.19.1 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.
- 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, 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.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.19.2 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: ADD_DICT, WRITE_DICT, REMOVE_DICT Built-In Procedures. Refer to the program example for the DISCTRL_LIST Built-In Procedure. Chapter 9 "DICTIONARIES AND FORMS"
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.19.3 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 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: WRITE_DICT_V Built-In Procedure, Chapter 9 "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.

Example A.19.3 READ_DICT_V Built-In Procedure

```
--------------------------------------------
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
```

A.19.4 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
[out] 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] stat us : 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 pane ID before you open the keyboard file that is associated with READ_KB. The pane ID for the iPendant can be either 1, 2 or 3.
- The application running in Pane ID 1 is stored in $TP_CURSCRN. Pane ID 2 is stored in $UI_CURSCRN[1], or in general $UI_CURSCRN[device_stat-1]. The CRT application 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 (CRT/KB's) 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 input. 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: The following example suspends any teach pendant reads, uses READ_KB to read a single key, and then resumes any suspended reads.

Example A.19.4 READ_KB Built-In Procedure

```
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
```

A.19.5 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.19.5):
    - Arithmetic operators (+, +, *, /)
    - Relational operators (>, >=, =, < >, <, <=)

Table A.19.5 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.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.19.6 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.

A.19.7 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 11, ``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 11, ``Input/Output System,'' Appendix D, ``Syntax Diagrams,'' for more syntax information
Example: In the following example, the robot hand, specified by **gripper** , is relaxed using the RELAX HAND statement.

Example A.19.7 RELAX HAND Statement

```
PROGRAM relaxhnd
%NOLOCKGROUP
%NOPAUSESHFT
VAR
 gripper : INTEGER
BEGIN
 gripper = 1
 RELAX HAND gripper
 WAIT FOR DIN[1]=ON
 CLOSE HAND gripper
END relaxhnd
```

A.19.8 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: ADD_DICT Built-In Procedure, Chapter 9 "DICTIONARIES AND FORMS"
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.19.9 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: COPY_FILE, DELETE_FILE Built-In Procedures

A.19.10 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.

⚠CAUTION
System may be down if you rename the variable of system KAREL program by mistake. Do not use except when absolutely necessary.

See Also: CREATE_VAR, SET_VAR Built-In Procedures

A.19.11 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.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.8 , "Manipulating Values of Dynamically Displayed Variables" (CHG_DATA.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.19.12 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.

A.19.13 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.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.19.14 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 >= 0, let n be a positive INTEGER such that n <= x <= n + 1
  - If x >= n + 0.5, then n + 1 is returned; otherwise, n is returned.
  - If x <= 0, let n be a negative INTEGER such that n >= x >= n - 1
  - If x <= 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: TRUNC Built-In Function
Example: Following is to do ROUND and show the result.

```
PROGRAM round01
%NOLOCKGROUP
VAR
    int_value   : integer
    real_value : real
BEGIN
   WRITE ('input real value to round:',CR)
   READ (real_value,CR)
   int_value = round(real_value)
   WRITE ('rounded value:', int_value,CR)
END round01
```

A.19.15 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.
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 ,"Standard Routines" (ROUT_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.19.16 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.19.16 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.19.16 , 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".

⚠**WARNING**
Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: CONT_TASK, PAUSE_TASK, ABORT_TASK Built-In Procedures, Chapter 12 "MULTI-TASKING"
Example: Refer to Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL), for a detailed program example.

A.20 - S - KAREL LANGUAGE DESCRIPTION

A.20.1 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: CLEAR, LOAD Built-In Procedures
Example: Refer to Section B.1 ,"Saving Data to the Default Device" (SAVE_VRS.KL), for a detailed program example.

A.20.2 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.20.3 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.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.20.4 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: OPEN_TPE Built-in Procedure

Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.20.5 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: POST_SEMA, PEND_SEMA, CLEAR_SEMA Built-In Procedures, Chapter 12 "MULTI-TASKING"
Example: See examples in Chapter 12 "MULTI-TASKING"

A.20.6 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.20.7 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: ADD_BYNAMEPC, ADD_INTPC, ADD_REALPC, ADD_STRINGPC
Example: The following example sends event 12 to the PC with a data buffer.

Example A.20.7 SEND_DATAPC Built-In Procedure

```
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
```

A.20.8 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.

Example A.20.8 SEND_EVENTPC Built-In Procedure

```
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
```

A.20.9 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_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.20.10 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: DEF_WINDOW Built-In Procedure

Example: Refer to the following sections for detailed program examples:
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)

A.20.11 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 *posn* 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: SET_POS_REG, SET_JPOS_REG Built-In Procedures, GET_POS_REG, GET_JPOS_REG Built-In Functions
Example: The following example sets the extended position for the specified register.

Example A.20.11 SET_EPOS_REG Built-In Procedure

```
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
```

A.20.12 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.20.13 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
%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 should be used.
- *atr_value* is the parameter correspond to specified attribute.

See Also: SET_PORT_ATR Built-In Procedure, Subsection 7.3.1 "Setting File and Port Attributes".

A.20.14 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",

A.20.15 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: GET_INT_REG, GET_REAL_REG, SET_REAL_REG Built-in Procedures
Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL) , for a detailed program example.

A.20.16 SET_JPOS_REG Built-In Procedure

Purpose: Stores a JOINTPOS value in the specified position 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: GET_JPOS_REG, GET_POS_REG, SET_POS_REG, POS_REG_TYPE Built-in Procedures
Example: Refer to Section B.3 ,"Using Register Built-ins" (REG_EX.KL) , for a detailed program example.

A.20.17 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: GET_JPOS_TPE, GET_POS_TPE, SET_POS_TPE, GET_POS_TYP Built-in Procedures

Example: Refer to Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY.TP.KL), for a detailed program example.

A.20.18 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 9 "DICTIONARIES AND FORMS"
Example: Refer to Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL) , for a detailed program example.

A.20.19 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 beset to 1.

⚠WARNING

Do not run a KAREL program that performs motion if more than one motion group is defined on your controller. If your controller is set up for more than one motion group, all motion must be initiated from a teach pendant program. Otherwise, the robot could move unexpectedly and injure personnel or damage equipment.

See Also: The appropriate application-specific Operator's manual to setup Reference Positions
Example: In the following example, $REFPOS1[2].$perchpos and $REFPOS1[2].$perchtol are set according to **perch_pos** and **tolerance[i].**

Example A.20.19 SET_PERCH Built-In Procedure

```
PROGRAM SETPERCH
%NOLOCKGROUP
VAR
  perch_pos: JOINTPOS IN GROUP[1]
  idx         : INTEGER
  tolerance:ARRAY[6] OF REAL
BEGIN
  perch_pos = CURJPOS(0,0,1)
  FOR idx = 1 TO 6 DO
    tolerance[idx] = 0.01
  ENDFOR
  SET_PERCH (perch_pos, tolerance, 2)
END SETPERCH
```

A.20.20 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: Following is to delete assign from DOUT[1] to [48] and assign them to memory-image.

```
PROGRAM setpasg
%ALPHABETIZE
%NOLOCKGROUP
%INCLUDE KLIOTYPS

CONST
  SUCCESS = 0
  UNASIGNED = 13007

VAR
  port_n : INTEGER
  STATUS :INTEGER
BEGIN
  -- Delete assign
  FOR port_n = 0 TO 5 DO
    SET_PORT_ASG(IO_DOUT, port_n*8+1, 0, 0, 0, 0, 0, STATUS)
    IF (STATUS <> SUCCESS) AND (STATUS <> UNASIGNED) THEN
      -- Failed to SET_PORT_ASG
      WRITE ('SET_PORT_ASG built-in for DOUT (deletion) failed',CR)
      WRITE ('Status = ',STATUS,CR)
    ENDIF
  ENDFOR
  -- Assign as memory-image from DOUT[1] to [48]
  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
      WRITE ('SET_PORT_ASG built-in for DOUT (assignment) failed',CR)
      WRITE ('Status = ',STATUS,CR)
    ENDIF
  ENDFOR
  -- DOUT[1] is complemetary
  SET_PORT_MOD(io_dout, 1, 2, STATUS)
  IF (STATUS <> SUCCESS) THEN
    WRITE ('SET_PORT_MODE Failed on port ',1,CR)
    WRITE ('With Status = ',STATUS,CR)
  ENDIF
  -- DOUT[3] is polarity INVERSE
  SET_PORT_MOD(io_dout, 3, 1, STATUS)
  IF (STATUS <> SUCCESS) THEN
    WRITE ('SET_PORT_MODE Failed on port ',3,CR)
```

```
      WRITE ('With Status = ',STATUS,CR)
    ENDIF

    -- Set comment to DOUT[1]
    SET_PORT_CMT(IO_DOUT, 1, 'Equip-READY ',STATUS)
    IF (STATUS <> 0 ) THEN
      WRITE ('SET_PORT_CMT built-in failed',CR)
      WRITE ('Status = ',STATUS,CR)
    ENDIF
    -- Clear screen
    WRITE (CHR(128),CHR(137))
    -- USER screen
    FORCE_SPMENU(TP_PANEL,SPI_TPUSER,1)
    -- Show message
    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)
    ABORT
  END setpasg
```

A.20.21 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_sbits :Stop bits
  atr_dbits :Data length

atr_xonoff :XON/XOFF
atr_eol :End of line

- *atr_value* specifies the value for the attribute type. See Table A.20.21 on the following page which contains acceptable pre-defined attribute types with corresponding values.

Table A.20.21 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_sbits	sbits_1 sbits_15 sbits_2
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 Appendix D , "Character Codes"

- A returned integer is the status of this action to port.

See Also: SET_FILE_ATR Built-In Procedure, Subsection 7.3.1 , ``Setting File and Port Attributes," for more information
Example: Refer to the example for the GET_PORT_ATR Built_In Function.

A.20.22 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 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* is returned with zero if the parameters are valid and the specified comment can be set for the specified port.

See Also: SET_PORT_VALUE, SET_PORT_MOD, GET_PORT_CMT, GET_PORT_VALUE, GET_PORT_MOD Built-in Procedures
Example: Refer to SET_PORT_ASG Built-In Procedure example.

A.20.23 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. ports.
  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 SET_PORT_ASG Built-In Procedure example.

A.20.24 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: SET_PORT_ASG, GET_PORT_ASG, GET_PORT_SIM Built-in Procedures

A.20.25 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 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 (i.e. DOUT), 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 value can be set for the specified port.

See Also: SET_PORT_VALUE, SET_PORT_MOD, GET_PORT_CMT, GET_PORT_VALUE, GET_PORT_MOD Built-in Procedures
Example: The following example sets the value for a specified port.

Example A.20.25 SET_PORT_VAL Built-In Procedure

```
PROGRAM setpval
%ENVIRONMENT IOSETUP
%INCLUDE kliotyps

VAR
  stvl_stat:INTEGER
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
  -- Turn ON DOUT[5]
  stvl_stat = set_value(io_dout,5, ON)
END setpval
```

A.20.26 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.3 ,"Using Register Built-ins" (REG_EX.KL) , for a detailed program example.

A.20.27 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.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL), for a detailed program example.

A.20.28 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

A.20.29 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: SET_INT_REG, GET_REAL_REG, GET_INT_REG Built-in Procedures

A.20.30 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 register whose comment is being set.
- 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.20.31 SET_SREG_CMT Built-In Procedure

Purpose: To set the comment information of a KAREL string register based on a given string register number and a given comment.

NOTE
String register is available only on 7DA5 series or later.

Syntax: SET_SREG_CMT (register_no, comment_string, status)
Input/Output Parameters:
[in] register_no: INTEGER
[in] comment_string: STRING
[out] status: INTEGER
Details:

- Register_no specifies string register whose comment is being set.
- The comment_string represents the data which is to be used to set the comment of the given string register.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: GET_SREG_CMT, SET_STR_REG, SET_SREG_CMT Built-In Procedure

A.20.32 SET_STR_REG Built-In Procedure

Purpose: To set string to given string register.

> **NOTE**
> String register is available only on 7DA5 series or later.

Syntax: SET_STR_REG (register_no, value, status)
Input/Output Parameters:
[in] register_no: INTEGER
[in] value: STRING
[out] status: INTEGER
Details:

- Register_no specifies string register to set.
- The value represents the string data which is to be used to set the given string register.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.

See Also: GET_SREG_CMT, GET_STR_REG, SET_SREG_CMT Built-In Procedure

A.20.33 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.20.33 (a)–Bit INTEGER Format of Time

Bit	Comment
31-25	year
24-21	month
20-16	day
15-11	hour
10-5	minute
4-0	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: CNV_STR_TIME, GET_TIME Built-In Procedures
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** .

Example A.20.33 SET_TIME Built-In Procedure

```
WRITE('Enter the new time : ')
READ(str_time)
CNV_STR_TIME(str_time,int_time)
SET_TIME(int_time)
```

A.20.34 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 set a comment to. 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: GET_TPE_CMT and OPEN_TPE for more Built-in Procedures.

A.20.35 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.
- *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.20.36 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: GET_TSK_INFO Built-in Procedure
Example: See examples in Chapter 12 "MULTI-TASKING"

A.20.37 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: GET_ATTR_PRG Built-in Procedure

NOTE
The task name is usually started program name. Changing with SET_TSK_NAME makes it difficult to debug or causes some issues. Please do not use.

A.20.38 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 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* .
- *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.

Example: Following is to set the variable of other program with SET_VAR Built-In Procedure.

```
-- If DO[1] is ON, set [STVAR01A]int_val to 1.
-- If DO[1] is OFF, set [STVAR01A]int_val to 0.
PROGRAM STVAR01
%NOLOCKGROUP
CONST
   SUCCESS = 0
VAR
   entry    : INTEGER
   int_val : INTEGER
   status   : INTEGER
   prog_name: STRING[MAX_PROG_NAM]
BEGIN
   IF DOUT[1] = ON then
      prog_name = 'STVAR01A'
      int_val = 1
   ELSE
      prog_name = 'STVAR01B'
      int_val = 0
   ENDIF
   SET_VAR(entry, prog_name, 'int_val', int_val, status)
   if status <> SUCCESS then
```

```
      POST_ERR(status, '',0,0)
   ENDIF
END STVAR01
```

A.20.39 %SHADOWVARS Translator Directive

Purpose: This directive specifies that all variables by default are created in SHADOW.
Syntax : %SHADOWVARS

A.20.40 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.20.41 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: EVENT Condition
Example: Following is to detect DOUT[1] = ON with CONDITION[1]. EVENT[1] is occurred by its action.

```
PROGRAM SIG_EV
%NOLOCKGROUP

BEGIN
   DOUT[1] = OFF
   DOUT[3] = OFF
   CONDITION[1]:
      WHEN DOUT[1]=ON DO
      SIGNAL EVENT[1]
   ENDCONDITION

   CONDITION[2]:
      WHEN EVENT[1] DO
      DOUT[3] = ON
   ENDCONDITION

   ENABLE CONDITION[1]
   ENABLE CONDITION[2]

   WAIT FOR DOUT[3] = ON
   FORCE_SPMENU(TP_PANEL,SPI_TPUSER,1)
   WRITE('SIG_EV FINISHED',CR)
END SIG_EV
```

A.20.42 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, EVENT Condition
Example: Refer to the DISABLE CONDITION Statement example program.

A.20.43 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 12.7 , "Task Communication" for more information and examples.

A.20.44 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 ± 18000 degrees. Otherwise, the program will be aborted with an error.

Example: Refer to TAN Built-In Procedure for a detailed program example.

A.20.45 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.

Example A.20.45 SQRT Built-In Function

```
c = SQRT(a*a+b*b)
WRITE ('The hypotenuse of the triangle is ',c::6::2)
```

A.20.46 %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: Subsection 5.1.6 , ``Stack Usage,'' for information on computing stack size

A.20.47 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.20.47 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.1 ,"Saving Data to the Default Device" (SAVE_VR.KL)
Section B.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)
Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL)
Section B.10 , "Using the DISCTRL_ALPHA Built-in" (DCALP_EX.KL)
Section B.11 , "Applying Offsets to a Copied Teach Pendant Program" (CPY_TP.KL)

A.20.48 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.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL) for a detailed program example.

A.20.49 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)
  - FILE 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: Subsection 2.4.2 , ``User-Defined Data Structures"

A.20.50 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.5 , "Listing Files and Programs and Manipulating Strings" (LIST_EX.KL)
Section B.6 , "Using the File and Device Built-ins" (FILE_EX.KL)

A.21 - T - KAREL LANGUAGE DESCRIPTION

A.21.1 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 ± 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*).

Example A.21.1 TAN Built-In Function

```
PROGRAM ex_tan
%NOLOCKGROUP
VAR
    angle       : real
    ratio      : real
    sin_value : real
    cos_value : real

BEGIN
  WRITE ('enter an angle:')
  READ (angle,CR)

  sin_value = SIN(angle)
  cos_value = COS(angle)
  ratio = sin_value/cos_value
  WRITE ('Angle is ', angle,CR)
  WRITE ('Sin(angle) is ', sin_value, CR)
  WRITE ('Cos(angle) is ', cos_value, CR)
  IF ratio = TAN(angle) THEN
    WRITE ('ratio is correct',CR)
  ENDIF
END ex_tan
```

A.21.2 %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.21.3 %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.21.4 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)
- Basically translate KAREL program by ROBOGUIDE, not by TRASLATE Built-In.

Example: The following example program will create, translate, load, and run another program called hello.

Example A.21.4 TRANSLATE Built-In Procedure

```
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
```

A.21.5 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: 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.**

Example A.21.5 TRUNC Built-In Function

```
PROGRAM ex_trnc
%NOLOCKGROUP
VAR
    miles : real
    hours : real
    mph : integer
BEGIN
   WRITE ('enter miles driven, hours used: ')
   READ (miles,hours,CR)
   mph = TRUNC(miles/hours)
   WRITE ('miles per hour=',mph::5)
END ex_trnc
```

A.22 - U - KAREL LANGUAGE DESCRIPTION

A.22.1 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 or structure.

Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL) for a detailed program example.

A.22.2 %UNINITVARS Translator Directive

Purpose: This directive specifies that all variables are by default uninitialized.
Syntax : %UNINITVARS

A.22.3 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 PAUSE conditions.

See Also: PAUSE Actions

A.22.4 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* .

A.22.5 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.

A.23 - V - KAREL LANGUAGE DESCRIPTION

A.23.1 V_CAM_CALIB *i*RVision 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)
[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.23.1 .

Table A.23.1 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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

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('CAL1',1,STATUS)
    -- success if status is zero
  IF STATUS <> 0 THEN
    WRITE ('V_CAM_CALIB failed - ERROR ', STATUS, CR)
    ABORT
  ENDIF

END cal
```

A.23.2 V_FIND_VIEW *i*RVision Built-In Procedure

Purpose: Performs vision detection with using the image stored in a specified image register.
Syntax:V_FIND_VIEW(vp_name, camera_view, imreg_num, status)
[in] vp_name: STRING
[in] camera_view: INTEGER
[in] imreg_num: INTEGER
[out] status: INTEGER
% ENVIRONMENT GROUP: CVIS
Details:

- *vp_name* is the name of the vision process.
- *camera_view* is the number of the camera view. Specify a number from 1 to 4.
- *imreg_num* is the image register number.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.
- Calling V_FIND_VIEW after V_SNAP_VIEW has an equivalent effect as calling V_RUN_FIND.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).
- See also the *i*RVision Operator's Manual for image registers.

A.23.3 V_FIND_VLINE *i*RVision Built-In Procedure

Purpose: Execute an Image To Points Vision Process. This is the dedicated built-in for Image To Points Vision Process. This outputs the view lines which 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 index number of a camera view. If the vision process is an Image To Points Vision Process, please specify 1.
- *imreg_num* is the index number of the image register which stores an image. When you do not use the image register, that is, you want to snap and find, please specify 0 or negative integer.
- *vlines* is the PATH variable which 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 the 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 make KAREL programs, please make them include VITPTYPS.KL and define the PATH variable for the view lines by VLINES_T.

Example A.23.3(a) The type of the PATH variable for the view lines (VITPTYPS.KL)

```
%ENVIRONMENT vitpdef

TYPE
  VLINES_T = PATH NODEDATA = VITP_ND_VL_T, PATHHEADER = VITP_HD_VL_T
```

- 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.

Example A.23.3(b) The header and node of the PATH variable for the view lines

```
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
```

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.

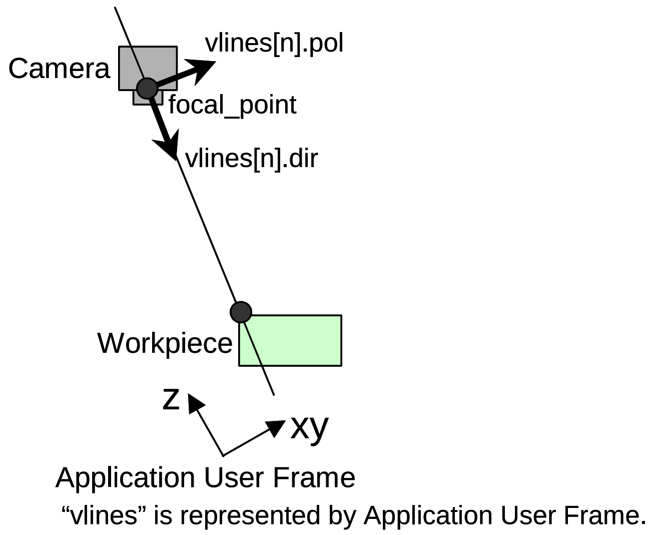


Fig. A.23.3 Physical relationship

- *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, INTSCT_VL_PL built-in waits for the completion of the processes of the vision process.
- Each time this built-in is executed, all the nodes of the PATH variable for the view lines are deleted and new nodes are added to the PATH variable.
- This built-in needs the ***i***RVision KAREL interface option (J870).
- You need the Virtual Robot with V7.70P/15 or later in order to translate this built-in.

See Also: INTSCT_VL_PL Built-In Procedure.

Example:

- The following is the KAREL program to get the view lines.

Example A.23.3(c) 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 a lot of 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 saves the DRAM variable contents to Flash ROM by using SAVE_DRAM built-in.

A.23.4 V_GET_OFFSET *i*RVision 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)
[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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

Example Program:

```
------------------------------------------------------------------
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.23.5 V_GET_PASSFL *i*RVision 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)

[in] vp_name : STRING

[in] register_no : INTEGER

[out] status : INTEGER

%ENVIRONMENT GROUP: CVIS

Details:

- *vp_name* is the name of the error proofing vision process created in setup mode
- *register_no* is the register number the command will set with the error proofing operations status.
  - Zero - error proofing operation failed
  - One - error proofing operation was successful
  - Two - error proofing 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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

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.23.6 V_OVERRIDE *i*RVision Built-In Procedure

Purpose: Assign a value to a specified Vision Override.
Syntax:
V_OVERRIDE(vo_name,value,status)
[in] vo_name: STRING
[in] value: REAL
[out] status: INTEGER
%ENVIRONMENT GROUP: CVIS
Details:
- *vo_name* is the name of the vision override.
- *value* is the value to be set.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.
- Vision Override allows you to override a parameter in a vision process before running. See also the *i*RVision Operator's Manual about Vision Override.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

A.23.7 V_RUN_FIND *i*RVision Built-In Procedure

Purpose: Starts an *i*RVision 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)
[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.

- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

Example Program:

```
-----------------------------------------------------------------
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
END vision
```

A.23.8 V_SET_REF *i*RVision 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)
[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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

A.23.9 V_SNAP_VIEW Built-In Procedure

Purpose: Snaps a new image and store it in a specified image register.
Syntax:V_SNAP_VIEW(vp_name, camera_view, imreg_num, status)
[in] vp_name: STRING
[in] camera_view: INTEGER
[in] imreg_num: INTEGER
[out] status: INTEGER
% ENVIRONMENT GROUP: CVIS
Details:

- *vp_name* is the name of the vision process.
- *camera_view* is the number of the camera view. Specify a number from 1 to 4.
- *imreg_num* is the image register number.
- *status* explains the status of the attempted operation. If not equal to 0, then an error occurred.
- Calling V_FIND_VIEW after V_SNAP_VIEW has an equivalent effect as calling V_RUN_FIND.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).
- See also the *i*RVision Operator's Manual for image registers.

A.23.10 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 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.23.10 Valid Data Types

Data Type	Value
POSITION	1
XYZWPR	2
XYZWPREXT	6
INTEGER	16
REAL	17
BOOLEAN	18
VECTOR	19
SHORT	23
BYTE	24
JOINTPOS1	25
CONFIG	28
FILE	29
PATH (for internal use)	31
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.**

Example A.23.10 VAR_INFO Built-In Procedure

```
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
```

A.23.11 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.23.11 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
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

- *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 and 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: FILE_LIST, PROG_LIST Built-In Procedures

A.23.12 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 (<>) 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:

Example A.23.12(a) VECTOR Data Type

```
VECTOR = STRUCTURE
 X:  REAL
 Y:  REAL
 Z:  REAL
ENDSTRUCTURE
Note:  All fields are read-write
```

Example: The following example shows VECTOR as variable declarations, as parameters in a routine, and as a function routine return type.

Example A.23.12(b) VECTOR Data Type

```
VAR
  direction, offset : VECTOR
ROUTINE calc_offset(offset_vec:VECTOR):VECTOR FROM util_prog
```

A.23.13 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:

- *devices* 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: FR: returns all three parameters, but the volume name is not supported. The FROM disk must be mounted in order to query it.
  FROM: 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: Subsection 1.4.1 , Status Memory in the "Status Displays and Indicators," chapter of the appropriate application-specific Operator's Manual .
Example: The following example gets information about the different devices.

Example A.23.13 VOL_SPACE Built-In Procedure

```
PROGRAM space
%NOLOCKGROUP
%ENVIRONMENT FLBT
VAR
  total:  INTEGER
  free:   INTEGER
  volume: STRING [30]
BEGIN
  VOL_SPACE('rd:', total, free, volume)
  VOL_SPACE('from:', 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
```

A.23.14 VREG_FND_POS *i*RVision 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)
[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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

A.23.15 VREG_OFFSET *i*RVision 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)
[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.
- This built-in requires the *i*RVision KAREL Interface Software Option (J870).

A.23.16 VT_ACK_QUEUE *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.17 VT_CLR_QUEUE *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.18 VT_DELETE_PQ *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).
- 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.

Example A.23.18 KAREL program to delete a part

```
----------------------------------------------------------------
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.23.19 VT_GET_AREID *i*RVision 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
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.20 VT_GET_FOUND *i*RVision 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)
[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 *i*RVision KAREL interface option (J870).

A.23.21 VT_GET_LINID *i*RVision 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
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.22 VT_GET_PFRT *i*RVision 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_consct, model_id, pfrt, status)
[in] area_num: INTEGER
[in] consecutive: INTEGER
[in] num_consct: 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_consct. 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 Tolerane" 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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.23 VT_GET_QUEUE *i*RVision 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)
[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 Subsection "Pick Program" in Chapter 4, “SETUP” of "R-30*i*A/R-30*i*A Mate CONTROLLER *i*RVision Visual Tracking START-UP GUIDANCE (B-82774EN-2)."
- 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 Chapter 5, "VARIARION" of "R-30*i*A/R-30*i*A Mate CONTROLLER *i*RVision Visual Tracking START-UP GUIDANCE (B-82774EN-2)."
- 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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.24 VT_GET_TIME *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.25 VT_GET_TRYID *i*RVision 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
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.26 VT_PUT_QUE2 *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

⚠ **CAUTION**
This built-in cannot push a tray.

A.23.27 VT_PUT_QUEUE *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.28 VT_READ_PQ *i*RVision 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)
[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 Robot folder of the ROBOGUIDE workcell by doing "Build" for a KAREL source. You cannot use another type for the PATH variable.

Example A.23.28(a) The type of the PATH variable for the information of parts (VSTKTYPS.KL)

```
%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.

Example A.23.28(b) The header and node of the PATH variable for the information of parts

```
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 $TRAY_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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).
- 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.

Example A.23.28(c) KAREL program to make a PATH variable

```
-----------------------------------------------------------------
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 saves 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 VSTKTYPS.KL as follows. Otherwise, you cannot translate that KAREL program.

Example A.23.28(d) The type of the PATH variable for the information of parts

```
%ENVIRONMENT vstkdef

TYPE
  VTPARTQ_T FROM makepq = PATH NODEDATA = VTPARTNODE_T, PATHHEADER = VTPARTHEADER_T
```

- Read the information of parts from a queue.

Example A.23.28(e) KAREL program to read the part information

```
----------------------------------------------------------------------
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.23.29 VT_SET_FLAG *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.30 VT_SET_LDBAL *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).

A.23.31 VT_WRITE_PQ *i*RVision 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)
[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 *i*RVision KAREL interface option (J870) and the *i*RVision Tracking Queue option (J874).
- 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.

Example A.23.31(a) KAREL program to write the information of a part

```
----------------------------------------------------------------------
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.24 - W - KAREL LANGUAGE DESCRIPTION

A.24.1 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 HANDLER" , Appendix E , ``Syntax Diagrams," for more syntax information

Example A.24.1 WAIT FOR STATEMENT

```
PROGRAM WAIT01
%NOLOCKGROUP
VAR
   done :BOOLEAN
BEGIN
   done = FALSE
   WAIT FOR done = TRUE
END WAIT01
```

A.24.2 WHEN Clause

Purpose: Used to specify a conditions/actions pair in a local or 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 statement can include multiple WHEN clauses.

See Also: Chapter 6 "CONDITION HANDLER" , Appendix E , ``Syntax Diagrams," for more syntax information
Example: Refer to the following sections for detailed program examples:
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)
Section B.7 , "Using Dynamic Display Built-ins" (DYN_DISP.KL)

A.24.3 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

A.24.4 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 : move_sys_var = value or move_sys_var[n] = value
move_sys_var : one of the system variables available for use in the WITH clause
n : specifies the group number
value : an expression of the type corresponding to the type of the system variable
Details:

- INTEGER values can be used where REAL values are expected.
- $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.

See Also: Chapter 6 "CONDITION HANDLER"

A.24.5 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.
- 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: 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.24.6 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: READ_DICT, REMOVE_DICT Built-In Procedures, Chapter 9 "DICTIONARIES AND FORMS"
Example: Refer to Section B.9 , "Displaying a List From a Dictionary File" (DCLST_EX.KL), for a detailed program example.

A.24.7 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 field names, and/or 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: READ_DICT_V Built-In Procedure, Chapter 9 "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.

Example A.24.7 WRITE_DICT_V Built-In Procedure

```
-------------------------------------------
TPTASKEG.UTX
-------------------------------------------
$ "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
```

A.25 - X - KAREL LANGUAGE DESCRIPTION

A.25.1 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, 32 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:

Example A.25.1 XYZWPR Data Type

```
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.
```

Example: Refer to the following sections for detailed program examples:
Section B.3 ,"Using Register Built-ins" (REG_EX.KL)
Section B.4 , "Position Data Set and Condition Handlers Program" (PTH_MOVE.KL)

A.25.2 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, 44 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:

Example A.25.2 XYZWPREXT Data Type

```
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.
```

A.26 - Y - KAREL LANGUAGE DESCRIPTION

There are no KAREL descriptions beginning with "Y".

A.27 - 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 , for more detailed information on each of the KAREL built-ins and commands.
Table B(a) 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(a) KAREL Example Programs

Program Name	Program Function	Built-ins Used	Section to Refer
SAVE_VRS.KL	Saves data to the default device.	DELETE_FILE SAVE	Section B.1
ROUT_EX.KL	Contains standard routines that are used throughout the program examples.	CHR FORCE_SPMENU	Section B.2
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.3
PTH_MOVE.KL	Teaches and moves along a path. Also uses condition handlers.	CHR CNV_REL_JPOS PATH_LEN SET_CURSOR	Section B.4
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.5

Program Name	Program Function	Built-ins Used	Section to Refer
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.6
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	Section B.7
CHG_DATA.KL	Processes and changes values of dynamically displayed variables.		Section B.8
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	Section B.9
DCLISTEG.UTX	Dictionary file.	N/A	Section B.9.1
DCALP_EX.KL	Uses the DISCTRL_ALPHA Built-in.	ADD_DICT CHR DISCTRL_ALPH FORCE_SPEMU POST_ERR SET_CURSOR SET_LANG	Section B.10

Program Name	Program Function	Built-ins Used	Section to Refer
DCALPHEG.UTX	Dictionary file.	N/A	Section B.10.1
CPY_TP.KL	Applies offsets to copied teach pendant pro grams.	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	Section B.11

B.1 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.

Example B.1(a) Saving Data Program - Overview

```
-----------------------------------------------------------------------------
----   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
```

Example B.1(b) Saving Data Program - Declarations Section

```
---------------------------------------------------------------------------
----   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
---------------------------------------------------------------------------
```

Example B.1(c) Saving Data Program - Create File Spec

```
---------------------------------------------------------------------------
----   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, irregardless 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
```

Example B.1(d) Saving Data Program - Delete/Overwrite

```
----------------------------------------------------------------------------
----   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 MC: 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 builtin 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
```

B.2 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

Example B.2(a) Standard Routines - Overview

```
------------------------------------------------------------------------
----   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
```

Example B.2(b) Standard Routines - Declaration Section

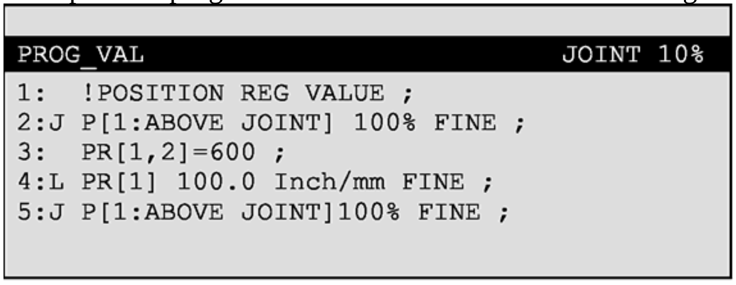
```
-----------------------------------------------------------------------
----   Section 1:  Program and Environment Declaration
-----------------------------------------------------------------------
PROGRAM ROUT_EX
%NOLOCKGROUP  ---- Don't lock any motion groups
%COMMENT = 'MISC_ROUTINES'
-----------------------------------------------------------------------
----   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
```

Example B.2(c) Standard Routines - Clears Screen and Displays Menu

```
-----------------------------------------------------------------------
----   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
```

B.3 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.

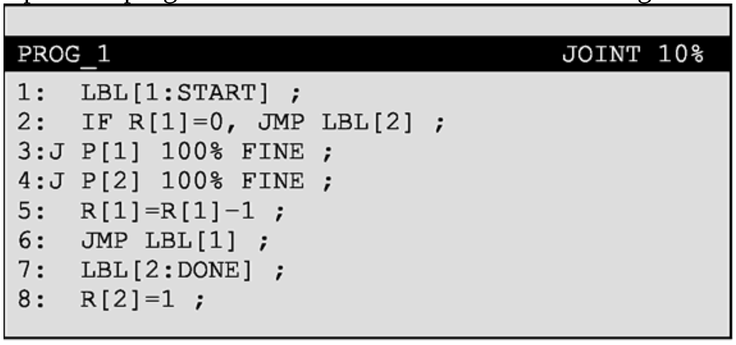


```
PROG_VAL                              JOINT 10%
1:   !POSITION REG VALUE ;
2:J  P[1:ABOVE JOINT] 100% FINE ;
3:   PR[1,2]=600 ;
4:L  PR[1] 100.0 Inch/mm FINE ;
5:J  P[1:ABOVE JOINT]100% FINE ;
```

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%
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 ;
```

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.

Example B.3(a) Using Register Built-ins Program - Overview

```
---------------------------------------------------------------------------
----   REG_EX.Kl
---------------------------------------------------------------------------
---- 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
```

Example B.3(b) Using Register Built-ins Program - Declaration Section

```
---------------------------------------------------------------------------
----   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
```

Example B.3(c) Using Register Built-ins - Storing and Manipulating Positions

```
-----------------------------------------------------------------------------
---- 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_indx, 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 builtin.
   -- 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]
```

Example B.3(d) Using Register Built-ins - Executing Program and Checking Register

```
-------------------------------------------------------------------
----   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_indx, FALSE)
  --Execute PROG_1.TPstarting 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
              WRITE ('PROG_1 failed to set R[2]',cr)-- by 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
    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
```

B.4 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.

Example B.4(a) Position Data Set and Condition Handlers Program - Overview

```
----------------------------------------------------------------------
----   PTH_MOVE.Kl
----------------------------------------------------------------------
----------------------------------------------------------------------
----   Section 0:  Detail about PTH_MOVE.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
----          PATH                            Sec 2
----          XYZWPR                          Sec 2
----   Directives:
----          ALPHABETIZE                      Sec 1
----          COMMENT                         Sec 1
----          ENVIRONMENT                      Sec 1
----   Built-in Functions & Procedures:
----          PATH_LEN                        Sec 4-C
----          CHR                             Sec 3-B; 4-B,D
----          CNV_REL_JPOS                     Sec 4-D
----          SET_CURSOR                       Sec 4-B
```

Example B.4(b) Position Data Set and Condition Handlers Program - Overview Continued

```
----   Statements:
----          Abort                           Sec 4-C
----         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
```

Example B.4(c) Position Data Set and Condition Handlers Program - Declaration Section

```
---------------------------------------------------------------------------
-
----   Section 1:  Program and Environment Declaration
---------------------------------------------------------------------------
-
PROGRAM PTH_MOVE        -- Define the program name
%ALPHABETIZE           -- Create the variables in alphabetical order
%COMMENT    = 'PATH MOVES'
%ENVIRONMENT PATHOP     -- Necessary for PATH_LEN
%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
   node_ind          :INTEGER           -- Index used when moving along path
   loop_pth          :INTEGER           -- Used in a FOR loop counter
   prg_abrt          :BOOLEAN           -- Set when program is aborted
   pth1             :PATH
   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
```

Example B.4(d) Position Data Set and Condition Handlers Program - Declare Routines

```
---------------------------------------------------------------------------
----   Section 3:  Routine Declaration
---------------------------------------------------------------------------
----   Section 3-A:  TP_CLS Declaration
----    ROUTINE TP_CLS FROM ROUT_EX       -- ROUT_EX must also be loaded.
---------------------------------------------------------------------------
---------------------------------------------------------------------------
----   Section 3-B:  YES_NO Declaration
---             Display choices on the function line of the TP.
----             Asks for user response.
---             F1 key is monitored by the Global condition handler
----             [CH_F1] and the F2 is monitored here.
----             If F1 is pressed the program will abort.
---             But, if the F2 is pressed the program will continue.
---------------------------------------------------------------------------
ROUTINE YES_NO
BEGIN
  WRITE TPFUNC (CHR(137))              -- Home Cursor in Function window
  WRITE TPFUNC ('  ABORT  CONT')         -- Display Function key options
  WAIT FOR TPIN[131]                   -- Wait for user to respond to
                                     -- continue.  If the user presses
                                     -- F1 (abort) condition handler
                                     -- CH_ABORT will abort program.
  WRITE TPFUNC (CHR(137))              -- Home Cursor in Function window
  WRITE TPFUNC ('  ABORT',chr(129))       -- Redisplay just Abort option and
                                     -- clear rest of Function window
END YES_NO
```

Example B.4(e) Position Data Set and Condition Handlers Program - Declare Condition Handlers

```
-----------------------------------------------------------------------------
----   Section 4:  Main Program
-----------------------------------------------------------------------------
BEGIN   -- PTH_MOVE
-----------------------------------------------------------------------------
----   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
```

Example B.4(f) Position Data Set and Condition Handlers Program - Teach and Move Along Path

```
----------------------------------------------------------------------------
----   Section 4-C: Verify PATH variable, pth1, has been taught
----------------------------------------------------------------------------
-- Check the number of nodes in the path
IF PATH_LEN(pth1) = 0 THEN                -- Path is empty (has no nodes)
   WRITE ('You need to teach the path.',cr)  -- Display instructions to user
   WRITE ('before executing this program.',cr)
   WRITE ('Teach the PATH variable pth1', CR, ' and restart the program',cr)
   ABORT                               -- Simply ABORT the task
                                      -- do not continue since there
ENDIF                                  -- are no nodes to move to
----------------------------------------------------------------------------
----   Section 4-D: Creating a joint position and moving along paths
----------------------------------------------------------------------------
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.
```

Example B.4(g) Position Data Set and Condition Handlers Program - Move Along Path

```
    OPEN_TPE('PTH_SUB',TPE_RWACC, TPE_WRTREJ, open_id, STATUS)
    IF STATUS <> 0 THEN
      WRITE (CR,'Open PTH_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 PTH_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 PTH_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 PTH_SUB failed:',STATUS,CR)
      ABORT
    ENDIF
    CLOSE_TPE(open_id, STATUS)
    CALL_PROG('pth_sub', prog_index)
  ENDIF

  WRITE ('pth_move Successfully Completed',CR)
END PTH_MOVE
```

B.5 LISTING FILES AND PROGRAMS AND MANIPULATING STRINGS

This program displays the list of files on the MC: 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.

Example B.5(a) Listing Files and Programs and Manipulating Strings - Overview

```
-----------------------------------------------------------------------------
----   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
```

Example B.5(b) Listing Files and Programs and Manipulating Strings - Overview Continued

```
----   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:
----          MC:                    Sec 4-C
----   Basic Concepts:
----       Semi-colon(;) as statement separator
----       Nested IF..THEN..ELSE..IF..THEN..ELSE..ENDIF..ENDIF structure
----       Concatenation of STRINGS using '+'
----
```

Example B.5(c) Listing Files and Programs and Manipulating Strings - Declarations Section

```
-----------------------------------------------------------------------------
----   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 form PROG_LIST
     n_progs      :INTEGER   -- Number of programs returned from PROG_LIST
     status   :INTEGER  -- Status of built-in procedure call
```

Example B.5(d) Listing Files and Programs and Manipulating Strings - Declare Routines

```
-----------------------------------------------------------------------------
----   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
```

Example B.5(e) Listing Files and Programs and Manipulating Strings - Main Program

```
-----------------------------------------------------------------------------
----   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 ('Manupulating 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.
```

Example B.5(f) Listing Files and Programs and Manipulating Strings - Create and Load 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, initi)
  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
```

Example B.5(g) Listing Files and Programs and Manipulating Strings - List Programs

```
-----------------------------------------------------------------------------
---- Section 4-C: Check the file listing of the drive MC: and display them
-----------------------------------------------------------------------------
  --- Display a directory listing of files on the MC:
  file_spec = 'MC:*.*'      -- All files in MC: 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 builtin 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 builtin 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
```

B.6 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 MC: 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.

Example B.6(a) File and Device Built-ins Program - Overview

```
-----------------------------------------------------------------------------
----   FILE_EX.Kl
-----------------------------------------------------------------------------
----   Section 0:  Detail about FILE_EX.kl
-----------------------------------------------------------------------------
---- Elements of KAREL Language Covered:         In Section:
----   Action:
----   Clauses:
----          FROM                         Sec 3
----   Conditions:
----   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
```

Example B.6(b) File and Device Built-ins Program - Overview Continued

```
----    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:
----           MC                            Sec 4-B
----           MF3                             Sec 4-B
----           RD                              Sec 4-B
----           FR                              Sec 4-B
```

Example B.6(c) File and Device Built-ins Program - Declaration Section, Declare Routines

```
-----------------------------------------------------------------------------
----    Section 1:  Program and Environment Declaration
-----------------------------------------------------------------------------
PROGRAM FILE_EX
%nolockgroup
%comment = 'COPY FILES'
-----------------------------------------------------------------------------
----    Section 2:  Variable Declaration
-----------------------------------------------------------------------------
CONST
   SUCCESS    = 0         -- Success status from builtins
   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 FANUC Robotics Controller KAREL Setup and Operations Manual for an
Error Code listing
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
```

Example B.6(d)   File and Device Built-ins Program - Mount and Copy to RAM Disk

```
------------------------------------------------------------------------------
----   Section 4-B: Mount RAMDISK and start copying from MC to
MF3:
------------------------------------------------------------------------------
to_dev = 'MF3:'
REPEAT                          -- Until all files have been copied
  cpy_stat = FINISHED
  WRITE('COPYing......',cr)
  -- Copy the files from MC: to to_dev and overwrite the file if it
  -- already exists.
  COPY_FILE('MC:*.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
```

Example B.6(e) File and Device Built-ins Program - Mount and Copy to RAM Disk Continued

```
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
```

B.7 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 cancelled 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.

Example B.7(a) Using Dynamic Display Built-ins - Overview

```
-----------------------------------------------------------------------------
----   DYN_DISP.Kl
-----------------------------------------------------------------------------
----   Section 0:  Detail about DYN_DISP.KL
-----------------------------------------------------------------------------
---- Elements of KAREL Language Covered:       In Section:
----   Actions:
----   Clauses:
----             FROM                         Sec 3-C
----             IN CMOS                       Sec 2
----             WHEN                         Sec 4
----   Conditions:
----             ABORT                         Sec 4
----   Data types:
----             BOOLEAN                       Sec 2
----             INTEGER                       Sec 2
----             REAL                         Sec 2
----             STRING                        Sec 2
----   Directives:
----             ALPHABETIZE                    Sec 1
----             COMMENT                       Sec 1
----             NOLOCKGROUP                    Sec 1
```

Example B.7(b) Using Dynamic Display Built-ins - Overview Continued

```
----   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:
----            TPPROMPT                       Sec 4-B,C
----   Predefined Windows:
----            T_FU                          Sec 3-A,B
```

Example B.7(c) Using Dynamic Display Built-ins - Declaration Section

```
-----------------------------------------------------------------------------
----   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
  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.
```

Example B.7(d) Using Dynamic Display Built-ins - Declare Routines

```
-----------------------------------------------------------------------------
----   Section 3: Routine Declaration
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
----   Section 3-A: SET_INT Declaration
----               Set all the input parameters for the INI_DYN_DISI call.
-----------------------------------------------------------------------------
ROUTINE Set_Int
Begin
  -- Valid predefined windows are described in
  -- Chapter 7.10.1, "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
  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:
---             --------
---            |xxxxxxxx|
---             --------
---     Where the integer value  will be left justified.
---     The x's will be the integer value unless the integer value is
---     less then 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.
End Set_Int
```

Example B.7(e) Using Dynamic Display Built-ins - Declare Routines Continued

```
-----------------------------------------------------------------------------
----   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.
```

```
   Char_Size       = 0              -- Normal
   Row            = 2              -- Specify the location within 'TFU'
   Col            = 16             --    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 then 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.
End Set_Real
```

Example B.7(f) Using Dynamic Display Built-ins - Declare Routines Continued

```
-----------------------------------------------------------------------------
----    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 7.10.1, "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
   Char_Size       = 0             -- Normal
   Row            = 3             -- Specify the location within 'T_FU'
   Col            = 16            --    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:
---            --------
---           |xxxxxxxx|
---            --------
---      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 bool1.
End Set_Bool
```

Example B.7(g) Using Dynamic Display Built-ins - Declare Routines Continued

```
-----------------------------------------------------------------------------
----    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 7.10.1, "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
  Char_Size       = 0              -- Normal
  Row            = 4             -- Specify the location within 'T_FU'
  Col            = 16            --    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:
---            --------
---           |xxxxxxxx|
---            --------
---     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.
End Set_Enum
```

Example B.7(h) Using Dynamic Display Built-ins - Declare Routines Continued

```
------------------------------------------------------------------------------
----   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 7.10.1, "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
  Char_Size       = 0           -- Normal
  Row            = 5          -- Specify the location within 'T_FU'
  Col            = 16         --    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:
---            --------
---           |xxxxxxxx|
---            --------
---     Where the string will be left justified.
---     The x's will be either 'RELEASED' or 'PRESSED'.
---     The string will also be normal video.
End Set_Port
```

Example B.7(i) Using Dynamic Display Built-ins - Declare Routines Continued

```
---------------------------------------------------------------------------
----   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 7.10.1, "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 th predefined display window
  Field_Width       = 10             -- Use the minimum width neccessary
  Char_Size         = 0              -- Normal
  Row              = 6              -- Specify the location within 'T_FU'
  Col              = 16             --    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:
---            --------
---           |xxxxxxxx|
---            --------
---     Where the string value will be left justified.
---     The x's will be the string value.
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
```

Example B.7(j) Using Dynamic Display Built-ins - Initiate Dynamic Displays

```
---------------------------------------------------------------------------
----   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
  Strl      = ''
  -- 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
```

Example B.7(k) Using Dynamic Display Built-ins - Initiate Dynamic Displays

```
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(Real1,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
```

Example B.7(l) Using Dynamic Display Built-ins - Execute Subordinate Task

```
-----------------------------------------------------------------------------
----    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
```

Example B.7(m) Using Dynamic Display Built-ins - User Response Cancels Dynamic Displays

```
----------------------------------------------------------------------------
----    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 Intl
  IF Status <> 0 THEN                     -- Check the status
     WRITE(' CND_DYN_DISI failed, Status=',status,CR)
  ENDIF
  CNC_DYN_DISR(Real1,Rel_Wind,Status)      -- Cancel display of Real1
  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
```

B.8 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.

Example B.8(a) Manipulate Dynamically Displayed Variables - Overview

```
----------------------------------------------------------------------------
----   CHG_DATA.KL
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----   Section 0:  Detail about CHG_DATA.KL
----------------------------------------------------------------------------
---- Elements of KAREL Language Covered:        In Section:
----   Actions:
----   Clauses:
----             FROM                        Sec 2
----   Conditions:
----   Data types:
----             INTEGER                      Sec 2
----             REAL                        Sec 2
----
----   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'
```

Example B.8.(b) Manipulate Dynamically Displayed Variables - Declaration Section

```
-----------------------------------------------------------------------------
----   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
  Bool1 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
-----------------------------------------------------------------------------
```

Example B.8(c) Manipulate Dynamically Displayed Variables - Main Program

```
------------------------------------------------------------------------------
----    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
```

B.9 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.9.1. DCLST_EX.KL controls the placement of the cursor along with the action taken for each command.

NOTE
The use of DISCTRL_FORM is the preferred method of displaying information. DISCTRL_FORM will automatically take care of all the key inputs and is much easier to use. For more information, refer to Chapter 9.

Example B.9(a) Display List from Dictionary File - Overview

```
-----------------------------------------------------------------------------
----   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
```

Example B.9(b) Display List from Dictionary File - Overview Continued

```
----          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
----    Statements:
----          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
----    Reserve Words:
----          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
----    Predefined File Names:
----          TPDISPLAY                       Sec 4-D,G,H,I
----          TPFUNC                         Sec 4-D,H
----          TPPROMPT                        Sec 4-D,H,I
----          TPSTATUS                        Sec 4-D,I
----    Devices Used:
----          RD2U                           Sec 3-B
----    Predefined Windows:
----          ERR                            Sec 4-C
----          T_ST                           Sec 4-C
----          T_FU                           Sec 4-C
----          T_PR                           Sec 4-C
----          T_FR                           Sec 4-C
```

Example B.9(c) Display List from Dictionary File - Declaration Section

```
------------------------------------------------------------------------------
----    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 DICTRL_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
```

Example B.9(d) Display List from Dictionary File - Declare Routines

```
-----------------------------------------------------------------------------
----    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
```

Example B.9(e) Display List from Dictionary File - Declare Error 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
```

Example B.9(f) Display List from Dictionary File - Setup and Define Screen

```
-----------------------------------------------------------------------------
----   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
                                 -- 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 7.10.1 "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')
```

Example B.9(g) Display List from Dictionary File - Write Elements to the 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)
```

Example B.9(h) Display List from Dictionary File - Initialize Display Data

```
------------------------------------------------------------------------------
----   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
```

Example B.9(i)   Display List from Dictionary File - Control Cursor Movement

```
-----------------------------------------------------------------------------
----   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')
```

Example B.9(j) Display List from Dictionary File - Control Cursor Movement Continued

```
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
```

Example B.9(k) Display List from Dictionary File - Cleanup and Exit Program

```
-----------------------------------------------------------------------------
----   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 that were
    Chk_stat ('Detaching ERR window')
  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
```

B.9.1 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.9.

Example B.9.1 (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
```

B.10 USING THE DISCTRL_ALPHA BUILT-IN

This program shows three different ways to use the DISCTRL_ALPH built-in. The DISCTRL_ALPH built-in displays and controls alphanumeric string entry in a specified window. Refer to Appendix A , "KAREL Language Alphabetic 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 Example B.10(c) .
Method 2 allows a comment to be entered using the default value for the dictionary name. See Section 4-B in Example B.10(c) .
Method 3 uses a user specified dictionary name and element to enter a program name. See Section 4-C in Example B.10(d) .
This program also posts all errors to the controller.

Example B.10(a) Using the DISCTRL_ALPHA Built-in - Overview

```
-----------------------------------------------------------------------------
----   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_ALPH                      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
```

Example B.10(b) Using the DISCTRL_ALPHA Built-in - Declaration Section

```
----------------------------------------------------------------------------
----   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
----------------------------------------------------------------------------
```

Example B.10(c) Using the DISCTRL_ALPHA Built-in - Enter Data from Teach Pendant

```
----------------------------------------------------------------------------
----   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)
```

Example B.10(d) Using the DISCTRL_ALPHA Built-in - Enter Data from CRT/KB

```
-----------------------------------------------------------------------------
---- 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
  ENDIF                                -- must be loaded to continue.
  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
  ENDIF                                 -- error ('err') window.
  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
 END DCALP_EX
```

B.10.1 Dictionary Files

This ASCII dictionary file is used to write text to the specified screen. DCALPH_EG.UTX is also used with DCALP_EX.KL. For more information on DCALP_EX.KL, refer to Section B.10.

Example B.10.1 DCALPHEG.UTX Dictionary File

```
.KL DCALPH
$, alpha_prog
" PRG      MAIN     SUB      TEST    >"&new_line
" PROC     JOB      MACRO            >"&new_line
" TEST1    TEST2    TEST3    TEST4   >"
```

NOTE
The greater than symbol (>) in Example B.10.1 is a reminder to use the NEXT key to scroll through the multiple lines. Also notice that the &new_line appears only on the first two lines. This ensures that the lines will scroll correctly.

B.11 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.

Example B.11(a) Applying Offsets to Copied Teach Pendant Program - Overview

```
----------------------------------------------------------------------------
----   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
```

Example B.11(b) Applying Offsets to Copied Teach Pendant Program - Overview Continued

```
----   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
```

Example B.11(c) Applying Offsets to Copied Teach Pendant Program - Declaration Section

```
-----------------------------------------------------------------------------
----   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 builtin 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 as 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
```

Example B.11(d) Applying Offsets to Copied Teach Pendant Program - Declare Routines

```
-----------------------------------------------------------------------------
----    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
```

Example B.11(e) Applying Offsets to Copied Teach Pendant Program - Generate Program List for User

```
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 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
```

Example B.11(f) Applying Offsets to Copied Teach Pendant Program - Overwrite or Delete Program

```
-----------------------------------------------------------------------------
----   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 overwriten if it exists
     over_sw = TRUE
  ELSE             --Set over_sw so program is NOT overwriten if it exists
     over_sw = FALSE
  ENDIF
END GetOvrSw
```

Example B.11(g) Applying Offsets to Copied Teach Pendant Program - Input Offset Positions

```
-----------------------------------------------------------------------------
----   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
```

```
END GetOffset   -- is satisfied
```

Example B.11(h) Applying Offsets to Copied Teach Pendant Program - Apply Offsets to 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
 -- Get the available position number
 AVL_POS_NUM(open_id, num_pos, status)
 chk_stat(status) -- Check AVL_POS_NUM status
 WRITE('avl_pos_num status', status,cr)
 if status = SUCCESS then
   WRITE('num_pos:', num_pos, CR)
 endif
--- 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)
```

Example B.11(i) Applying Offsets to Copied Teach Pendant Program - Apply Offsets to Positions Cont.

```
--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
```

Example B.11(j) Applying Offsets to Copied Teach Pendant Program - Clears TP User Menu

```
-- 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
                              -- the teach pendant program
 ModifyPrg                     -- Modify the copied program by the offset
END CPY_TP
```

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 CHDIR command

Syntax: CHDIR < **device_name** >¥< **path_name** >¥ or CD < **device_name** >¥< **path_nam e** >
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.4 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.

NOTE
The programs for internal use are also deleted. Do not use.

Examples: KCL> CLEAR ALL
Are you sure? YES
KCL> CLEAR ALL Y

C.5 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.6 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.7 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.8 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.9 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.10 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 9 "DICTIONARIES AND FORMS"
Example: KCL> COMPRESS DICT tphcmneg

C.11 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 9 "DICTIONARIES AND FORMS"
Examples: KCL> COMPRESS FORM
KCL> COMPRESS FORM mnpalteg

C.12 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.13 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.14 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.15 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 commandor KCL> DIR ¥test_dir¥*.* display the subdirectory contents without using the CD command.

C.16 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.17 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.18 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.19 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.20 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.21 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.22 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.23 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.24 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 pc file 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 pc file 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.25 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 9 "DICTIONARIES AND FORMS"
Examples: KCL> LOAD DICT tpaleg tpal FRENCH
KCL> LOAD DICT tpaleg tpal

C.26 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 9 "DICTIONARIES AND FORMS".
Example: KCL> LOAD FORM
Loading FORM.TX with dictionary name FORM
Loading FORM.VR
KCL> LOAD FORM tpexameg
Loading TPEXAMEG.TX with dictionary name EXAM
Loading EXAM.VR

C.27 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.28 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.
Examples: KCL> LOAD PROGRAM test_prog
KCL> LOAD PROG

C.29 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.30 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.

C.31 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.32 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", and "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.33 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.34 MKDIR command

Syntax: MKDIR **<device_name>¥path_name**
where:
device_name : a valid storage device
path_name : a subdirectory to be created.

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.35 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.36 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.37 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.38 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.39 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.40 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.41 RESET command

Syntax: RESET
Purpose: Resets the error.
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.42 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.43 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.44 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 10.3 , ``Command Procedures,'' for more information
Examples: KCL> RUNCF startup output
KCL> RUNCF startup

C.45 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.46 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.47 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.48 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.49 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.50 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.51 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.52 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.
Example: KCL> SET CLOCK '02-JAN-xx 21:45'

C.53 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: Subsection 10.1.1 , ``Default Program''
Examples: KCL> SET DEFAULT test_prog
KCL> SET DEF test_prog

C.54 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.55 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.56 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: SIMULATE, UNSIMULATE command, Chapter 11 "INPUT/OUTPUT SYSTEM", application-specific FANUC Setup and Operations Manual.
Example: KCL> SET PORT DOUT [1] = ON
KCL> SET PORT GOUT [2] = 255
KCL> SET PORT AIN [1] = 1000

C.57 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.58 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: SHOW TRACE command

C.59 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.60 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.61 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.62 SHOW BUILTINS command

Syntax: SHOW BUILTINS
Purpose: Displays all the softpart built-ins that are loaded on the controller.
Example: KCL> SHOW BUILTINS

C.63 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.64 SHOW CLOCK command

Syntax: SHOW CLOCK
Purpose: Displays the current date and time of the controller clock.
See Also: SET CLOCK command
Example: KCL> SHOW CLOCK

C.65 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.66 SHOW DEFAULT command

Syntax: SHOW DEFAULT
Purpose: Shows the current default program name.
Example: KCL> SHOW DEFAULT

C.67 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.68 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.69 SHOW GROUP command

Syntax: SHOW GROUP
Purpose: Shows the default group number.
Example: KCL> SHOW GROUP

C.70 SHOW HISTORY command

Syntax: SHOW HISTORY
Purpose: Shows the nesting information of the routine calls.
Example: KCL> SHOW HIST

C.71 SHOW LANG command

Syntax: SHOW LANG
Purpose: Shows the language specified in the system variable $LANGUAGE.
Example: KCL> SHOW LANG

C.72 SHOW LANGS command

Syntax: SHOW LANGS
Purpose: Shows all language currently available in the system.
Example: KCL> SHOW LANGS

C.73 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.74 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.75 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.76 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: SHOW VARIABLE command
Examples: KCL> SHOW SYSTEM REAL VALUES
KCL> SH SYS

C.77 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.78 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.79 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.
See Also: SET TRACE command
Example: KCL> SHOW TRACE

C.80 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: SHOW VARS command, SHOW VARIABLE command
Examples: KCL> SHOW TYPES test_prog FIELDS
KCL> SH TYPES

C.81 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: SHOW VARS command, 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.82 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: SHOW VARIABLE command, SHOW SYSTEM command, SHOW TYPES command
Example: KCL> SHOW VARS test_prog VALUES
KCL> SH VARS

C.83 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: SHOW VARS command, SHOW VARIABLE command
Examples: KCL> SHOW REAL test_prog VALUES
KCL> SH INTEGER

C.84 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: UNSIMULATE command
Examples: KCL> SIMULATE DIN[17]
KCL> SIM DIN[1] = ON
KCL> SIM AIN[1] = 100

C.85 STEP OFF command

Syntax: STEP OFF
Purpose: Disables single stepping for the program in which it was enabled.
Example: KCL> STEP OFF

C.86 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.87 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.
Basically do not use TRANSLATE command. Translate on ROBOGUIDE.
Examples: KCL> TRANSLATE testprog DISPLAY LIST
KCL> TRAN

C.88 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.89 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, WDO, AIN, AOUT, GIN, GOUT
See Also: SIMULATE command
Examples: KCL> UNSIMULATE DIN[17]
KCL> UNSIM ALL

C.90 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 10.3 , ``Command Procedures"
Example: The following is an example of an executable command procedure:

```
> SET DEF testprog
> RUN -- execute program
> WAIT PAUSE
> SHOW CURPOS -- display position of TCP when program pauses
> CONTINUE
> WAIT DONE
```

D CHARACTER CODES

D.1 CHARACTER CODES

This appendix lists the ASCII numeric decimal codes and their corresponding ASCII and European characters as implemented on the KAREL system. The ASCII character set is the default character set for the KAREL system.

Table D.1(a) ASCII character codes

Decimal Code	Character Value	Decimal Code	Character Value	Decimal Code	Character Value	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
027	(ESC)	059	;	091	[	123	{
028	(FS)	060	<	092	¥	124	|

Decimal Code	Character Value	Decimal Code	Character Value	Decimal Code	Character Value	Decimal Code	Character Value
029	(GS)	061	=	093	]	125	}
030	(RS)	062	>	094	^	126	~
031	(US)	063	?	095	—	127	(DEL)

Table D.1(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.1(c) 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
3	51	177	USER KEY 5
4	52	178	USER KEY 6
5	53	185	FWD

Code	Value	Code	Value
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.1(d) European character codes

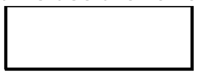
Code	Value	Code	Value	Code	Value
192	A`	213	O~	234	e^
193	A`	214	O:	235	e:
194	A^	215	OE	236	i`
195	A~	216	O/	237	i`
196	A:	217	U`	238	i^
197	Ao	218	U`	239	i:
198	AE	219	U^	240	
199	CC	220	U:	241	n~
200	E`	221	Y:	242	o`
201	E`	222		243	o`
202	E^	223	Bb	244	o^

Code	Value	Code	Value	Code	Value
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	

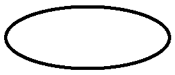
A^ = A with ^ on top
A` = A with ` on top
Ao = A with o on top
A~ = A with ~ on top
A: = A with .. on top
AE = A and E run together
OE = A and E run together
Bb = Beta

E SYNTAX DIAGRAMS

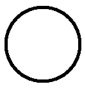
KAREL syntax diagrams use the following symbols:

- 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 

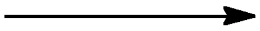
  A circle encloses special characters that are entered exactly as shown.

- Dot ●

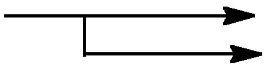
  A dot indicates a mandatory line–end (; or ENTER key) before the next syntax element.

- Caret ^

  A caret indicates an optional line–end.

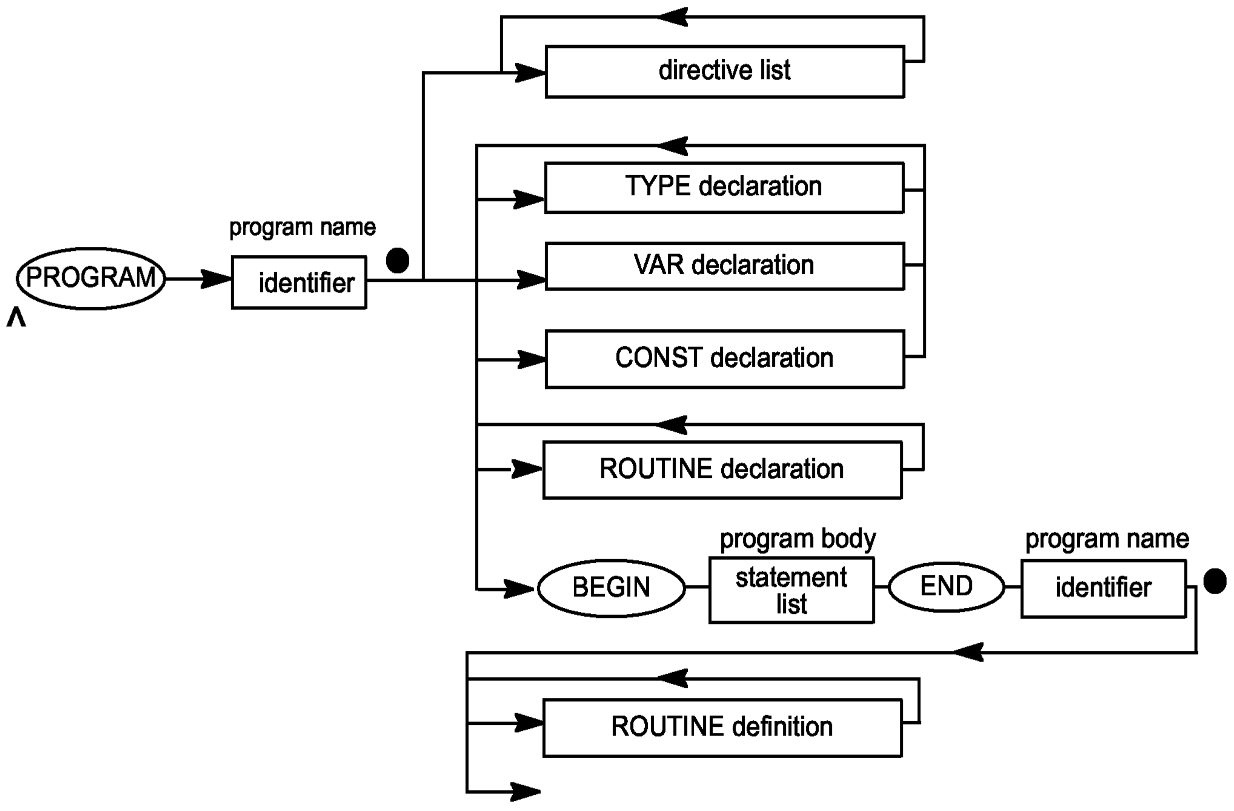
- Arrows 

  Arrows indicate allowed paths and the correct sequence in a diagram.

- Branch 

  Branches indicate optional paths or sequences.

PROGRAM– –module definition



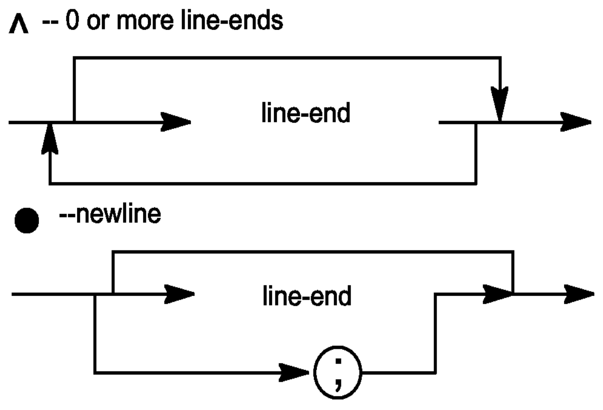


Fig. E(a)

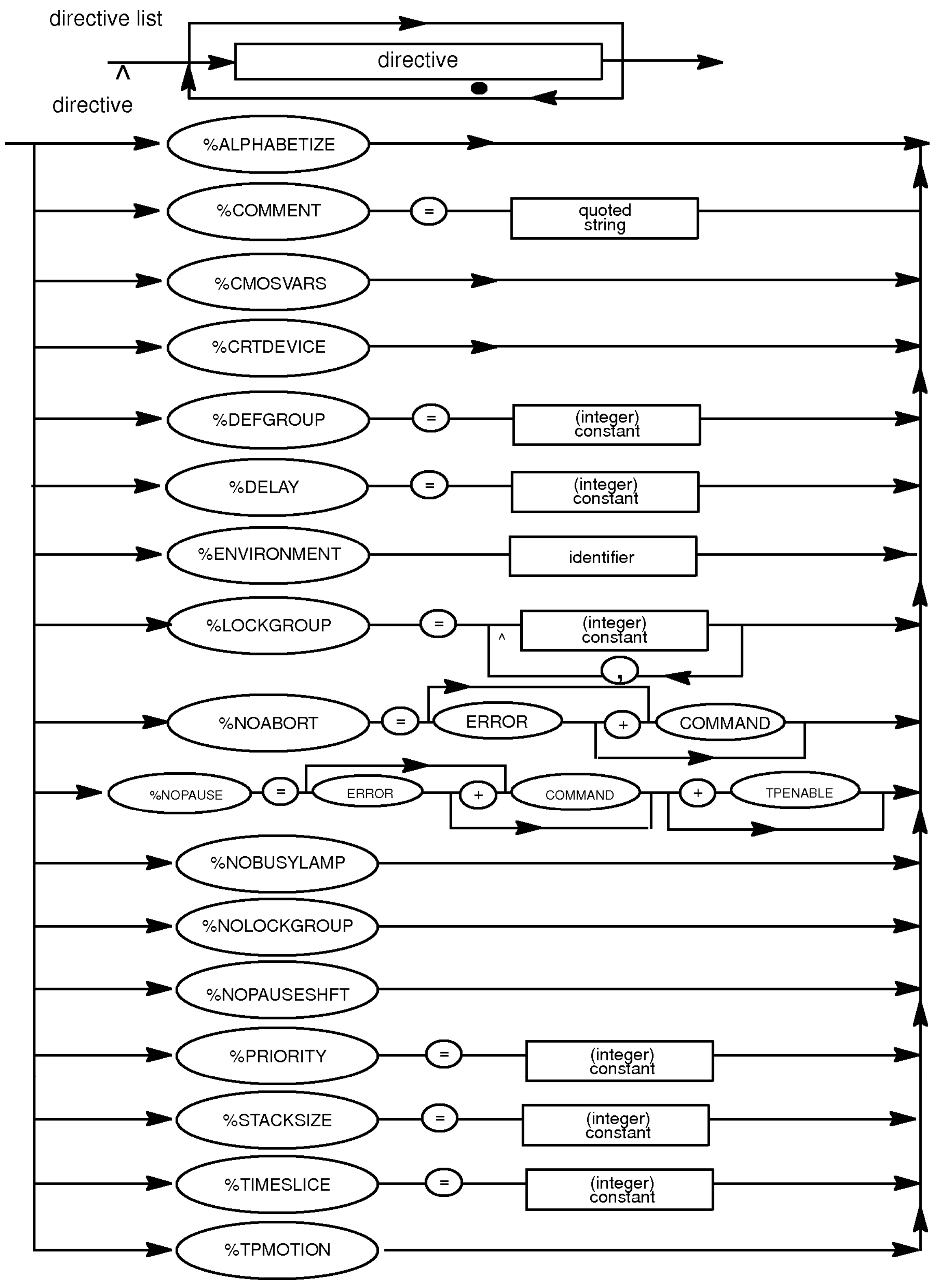
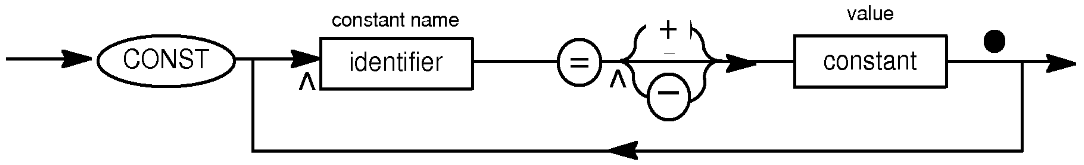
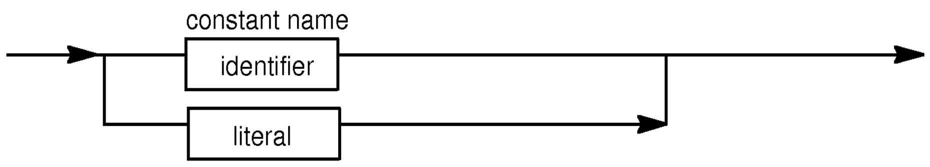

directive list
directive
directive
%ALPHABETIZE
%COMMENT
=
quoted string
%CMOSVARS
%CRTDEVICE
%DEFGROUP
(integer) constant
%DELAY
%ENVIRONMENT
identifier
%LOCKGROUP
,
%NOABORT
ERROR
+
COMMAND
%NOPAUSE
TPENABLE
%NOBUSYLAMP
%NOLOCKGROUP
%NOPAUSESHFT
%PRIORITY
%STACKSIZE
%TIMESLICE
%TPMOTION

Fig. E(b)

CONST --constant declaration



constant



TYPE -- type declaration

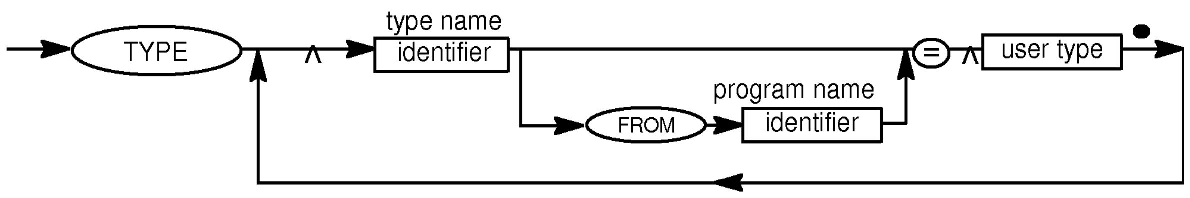
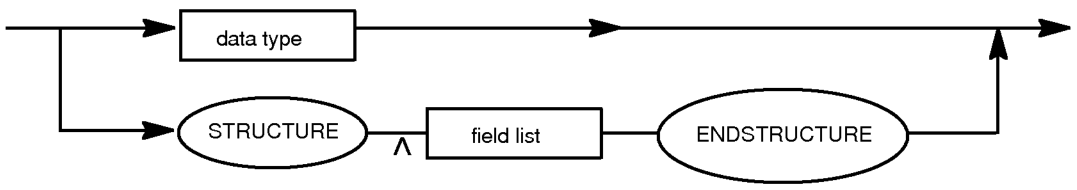
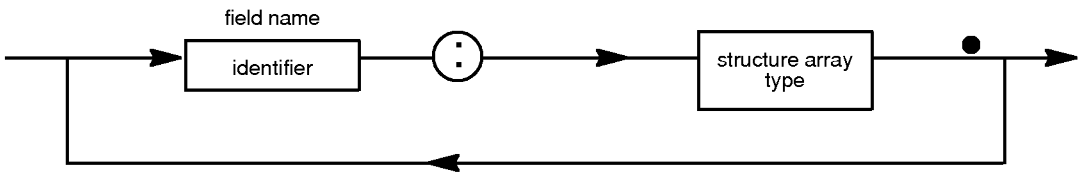


Fig. E(c)

user type



field list



structure array type

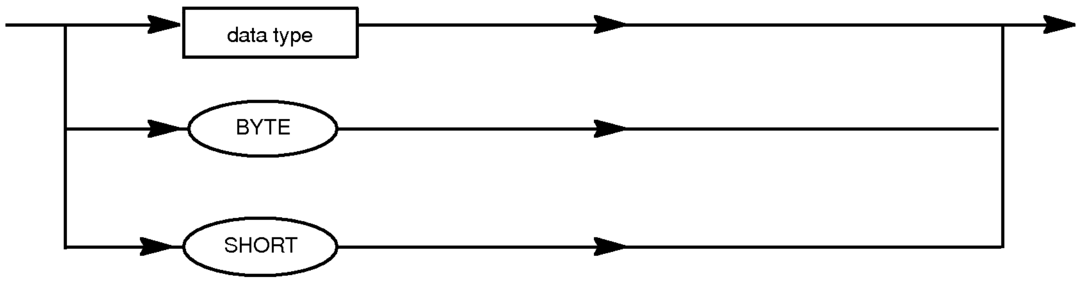


Fig. E(d)

data type

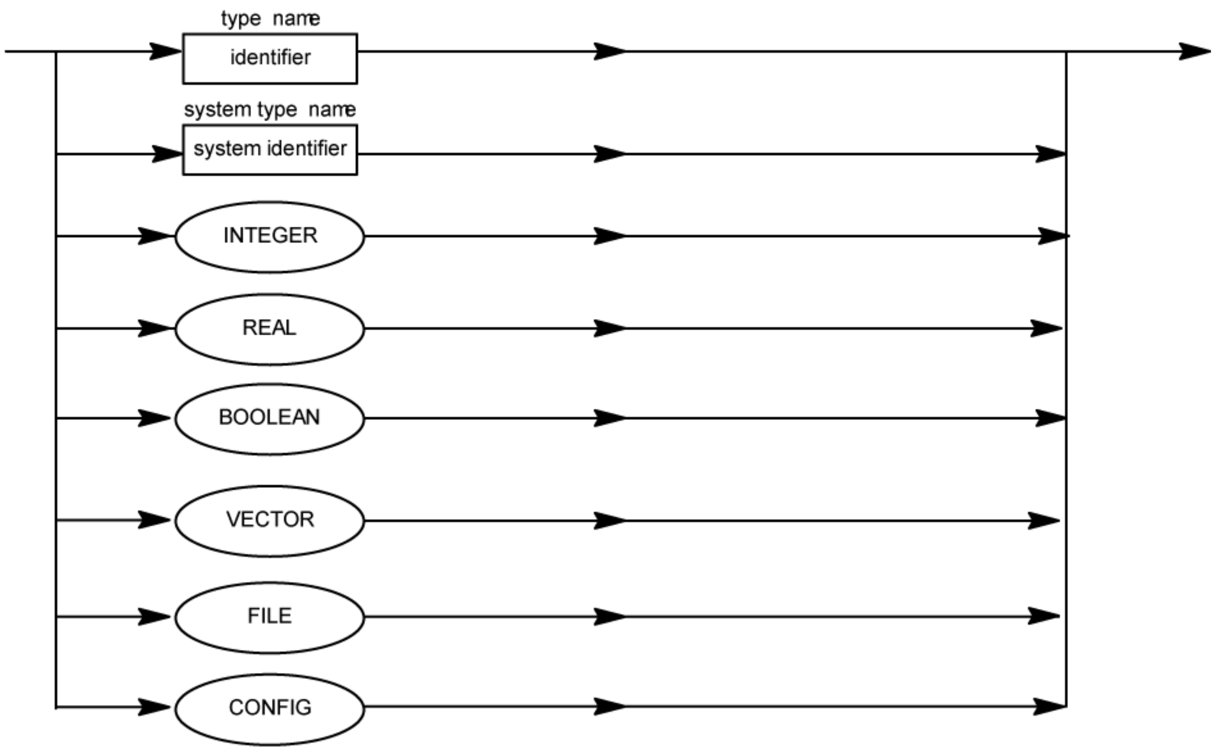


Fig. E(e)

data type continued

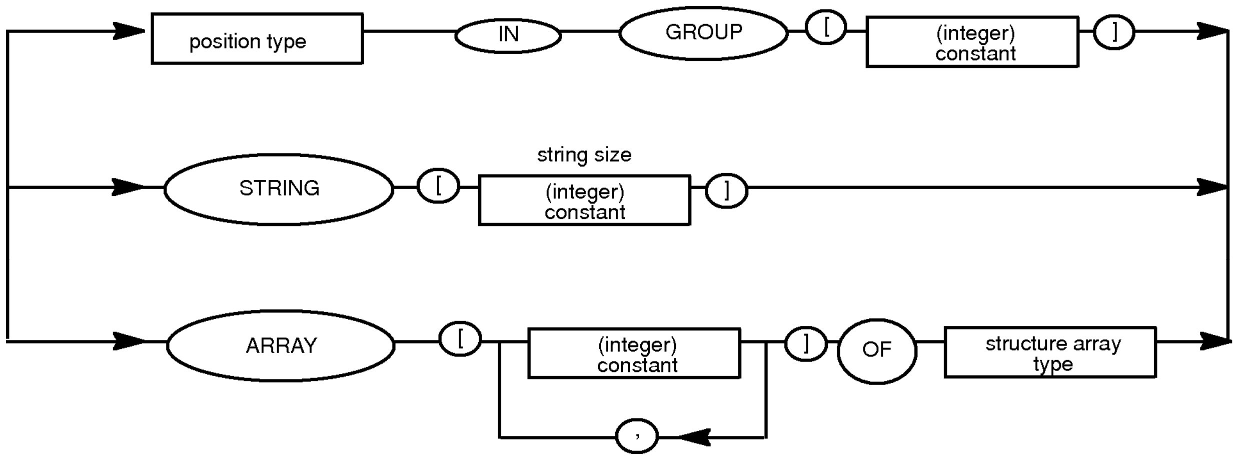


Fig. E(f)

position type

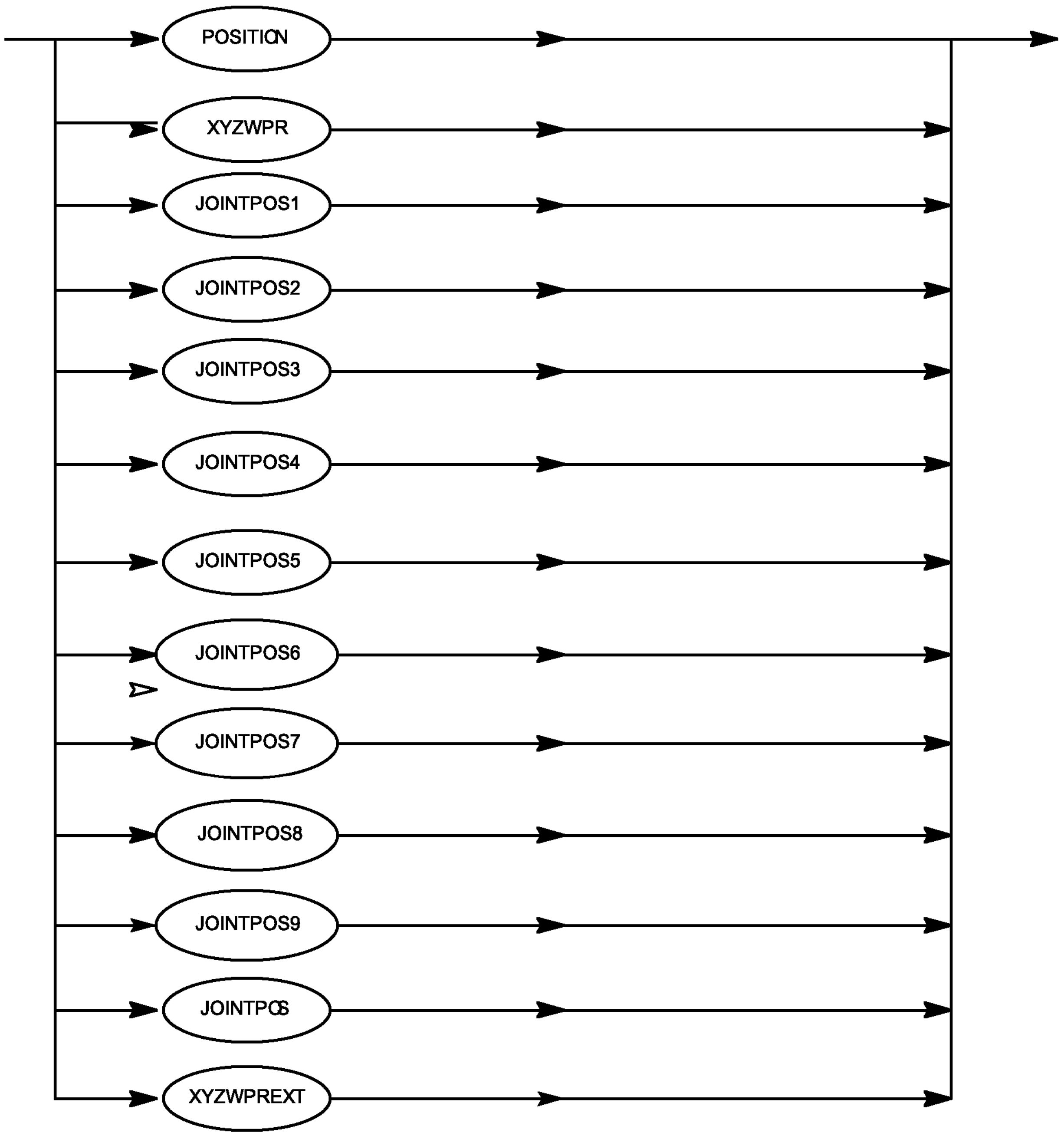
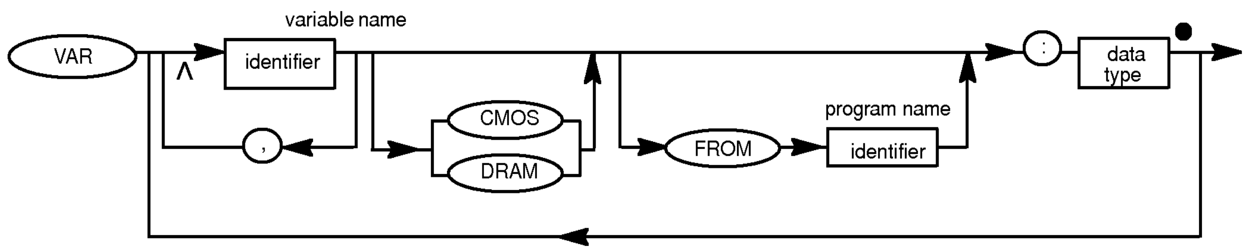


Fig. E(g)

VAR -- variable declaration



ROUTINE -- routine declaration

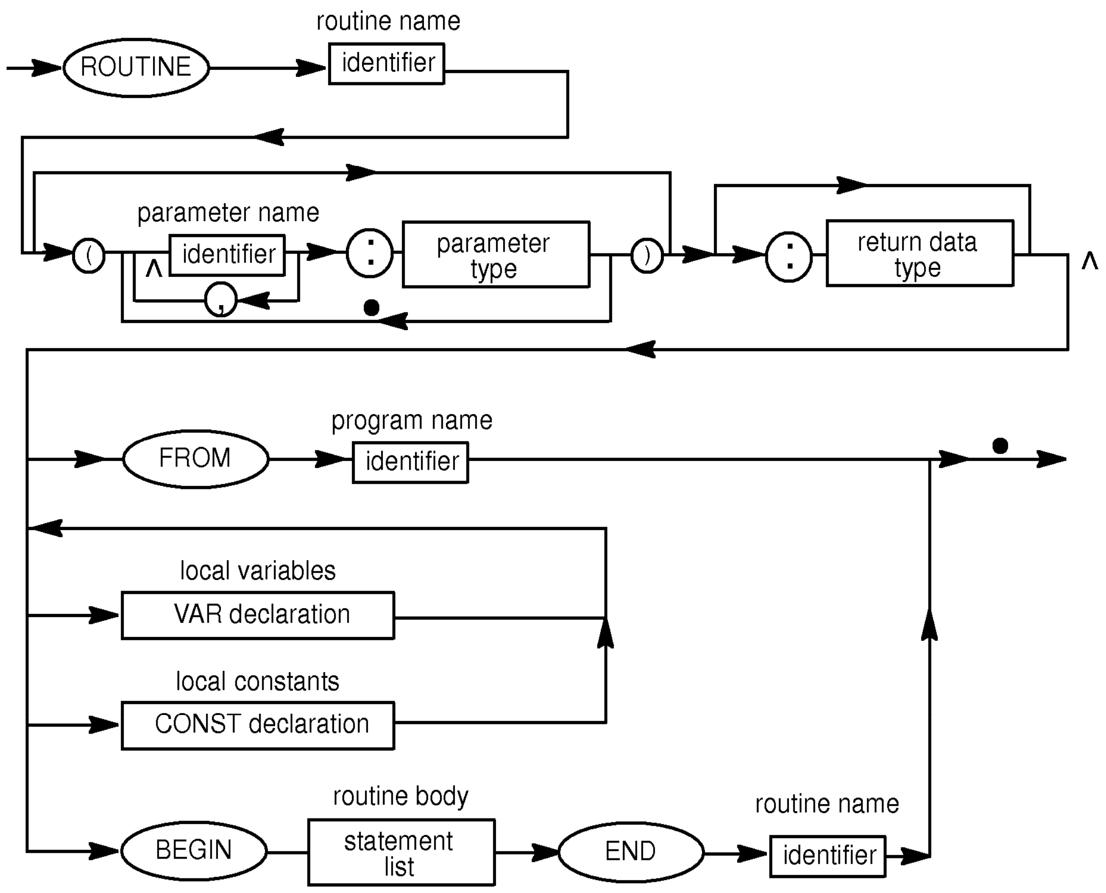


Fig. E(h)

return data type

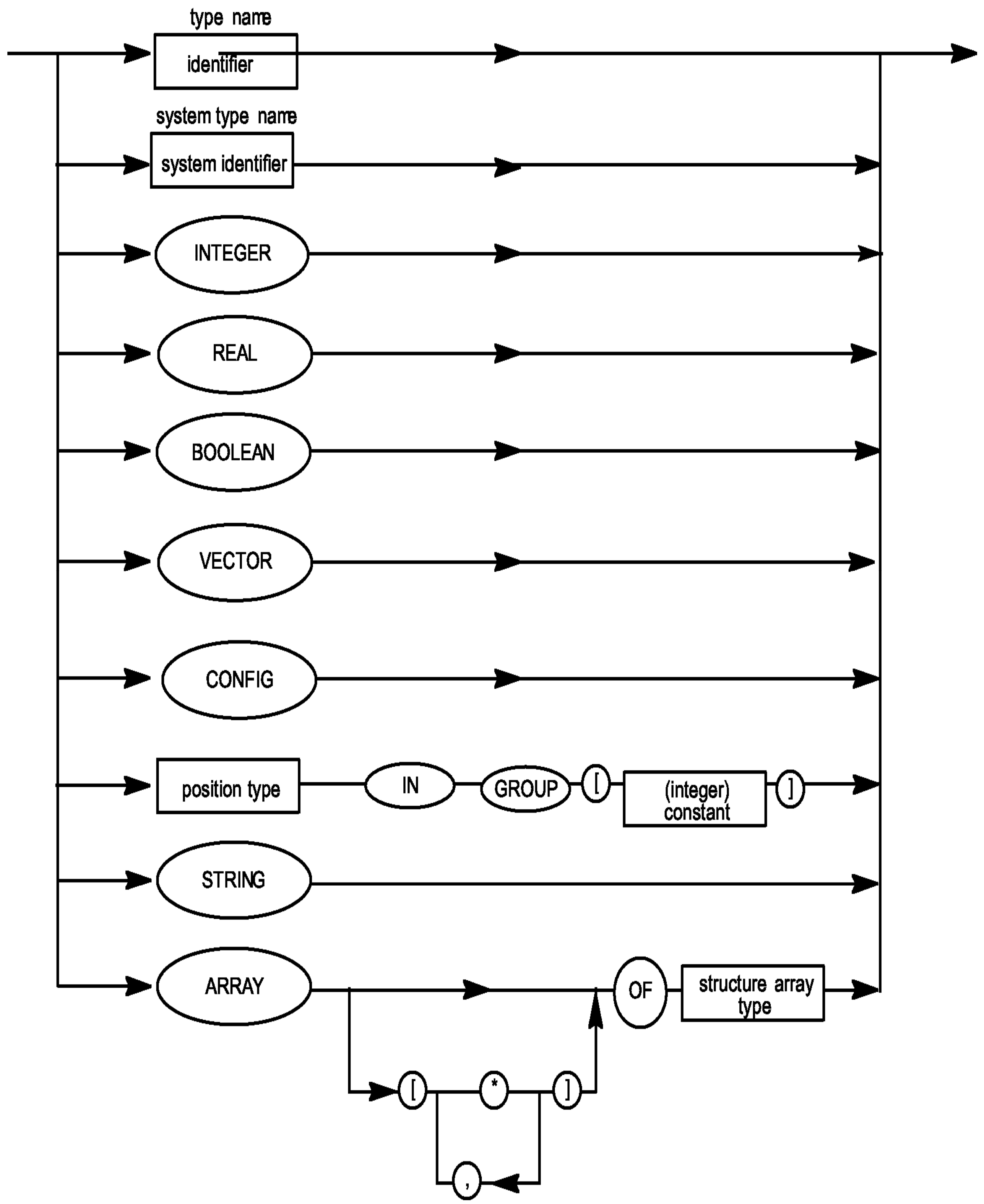


Fig. E(i)

parameter type

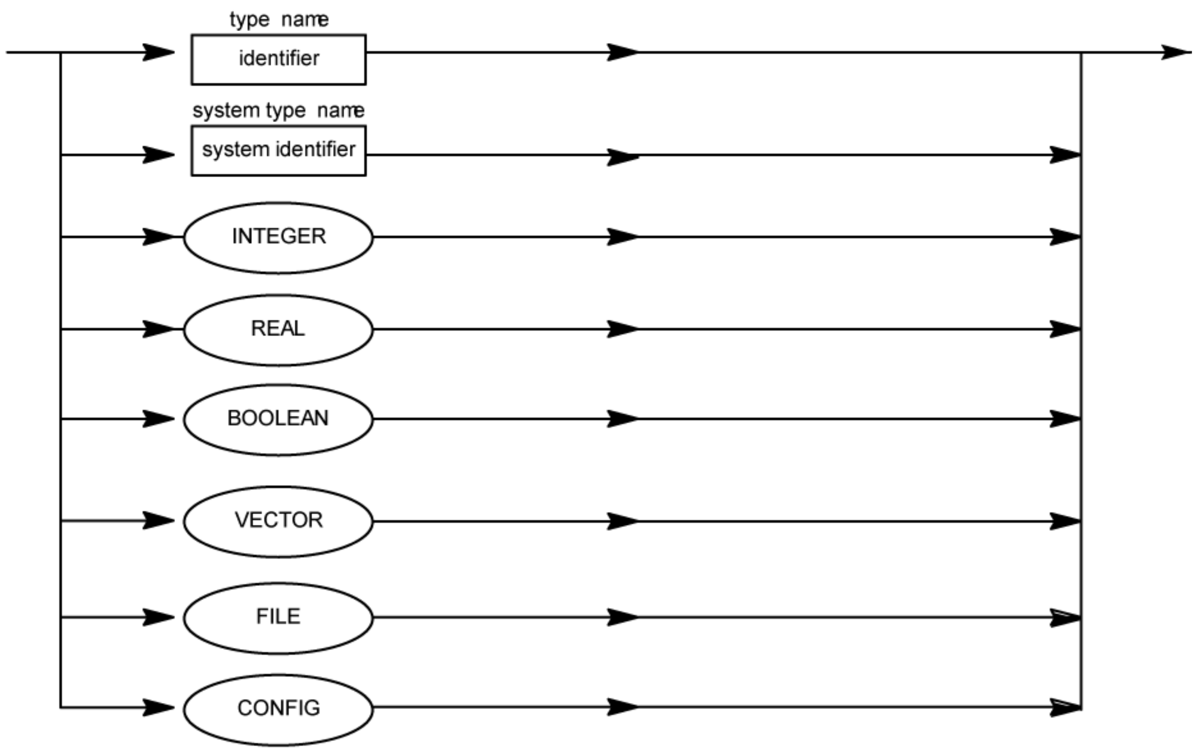


Fig. E(j)

parameter type continued

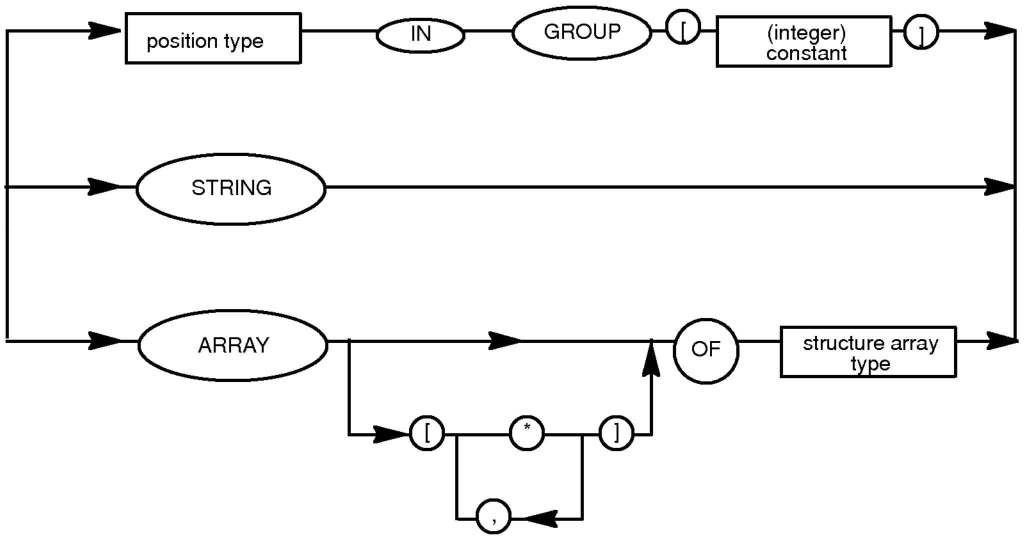
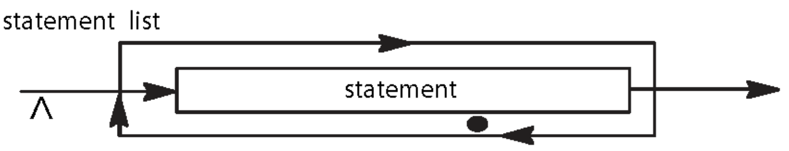


Fig. E(k)


statement list
statement
Λ
●

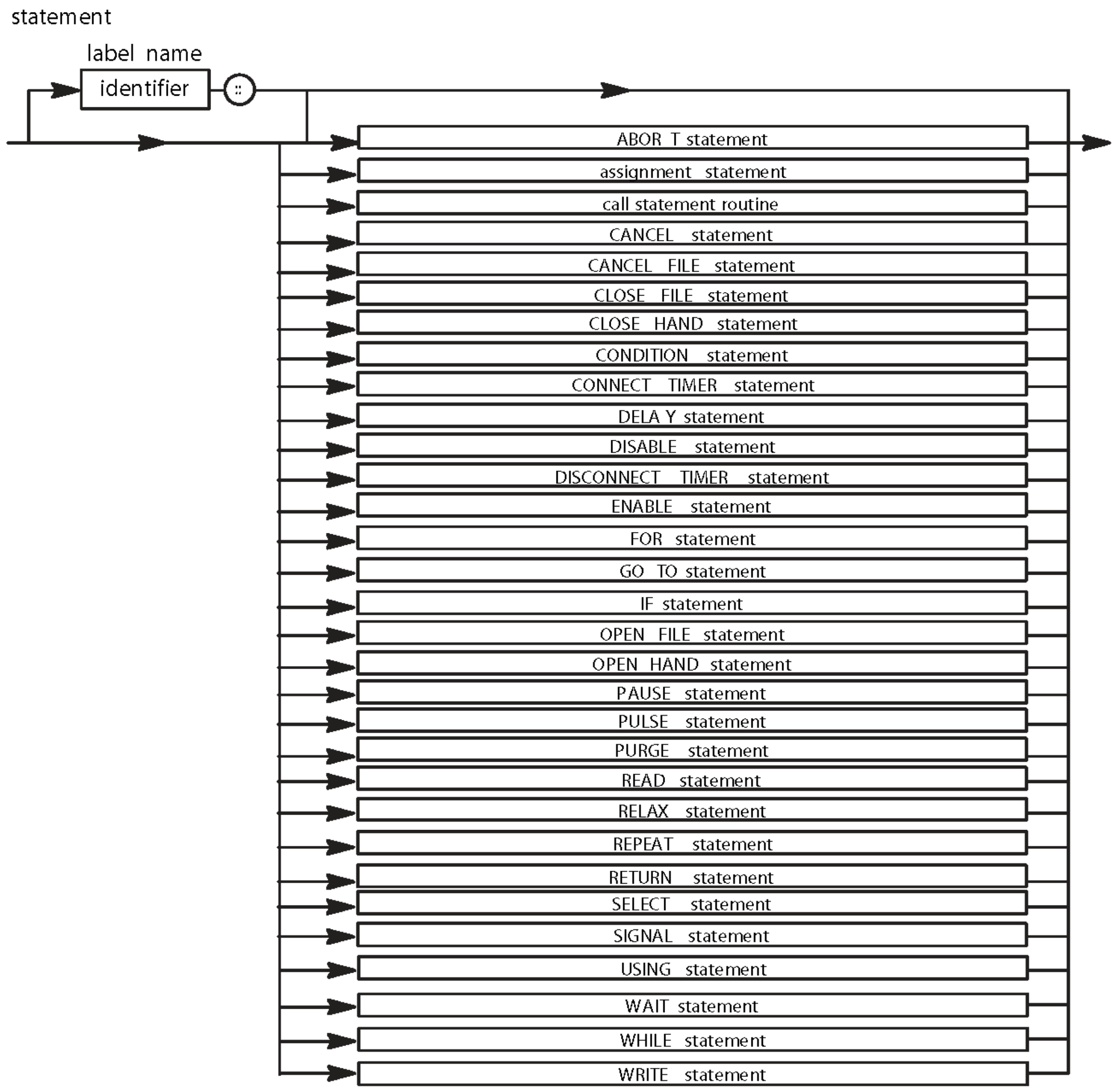
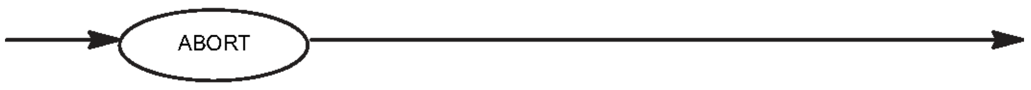
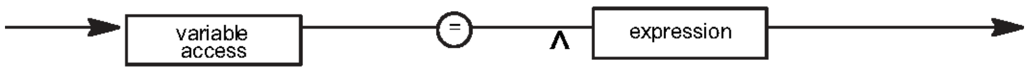

statement
label name
identifier
::
ABOR T statement
assignment statement
call statement routine
CANCEL statement
CANCEL FILE statement
CLOSE FILE statement
CLOSE HAND statement
CONDITION statement
CONNECT TIMER statement
DELA Y statement
DISABLE statement
DISCONNECT TIMER statement
ENABLE statement
FOR statement
GO TO statement
IF statement
OPEN FILE statement
OPEN HAND statement
PAUSE statement
PULSE statement
PURGE statement
READ statement
RELAX statement
REPEAT statement
RETURN statement
SELECT statement
SIGNAL statement
USING statement
WAIT statement
WHILE statement
WRITE statement

Fig. E(l)

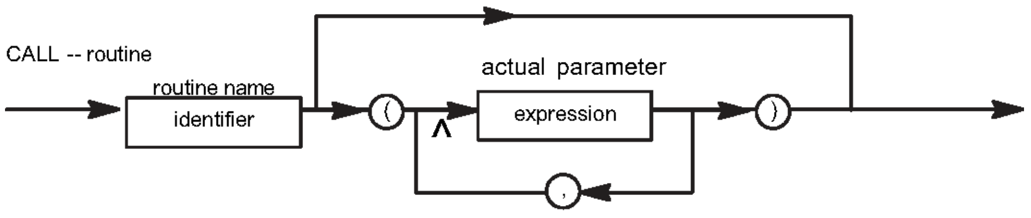
ABORT -- statement



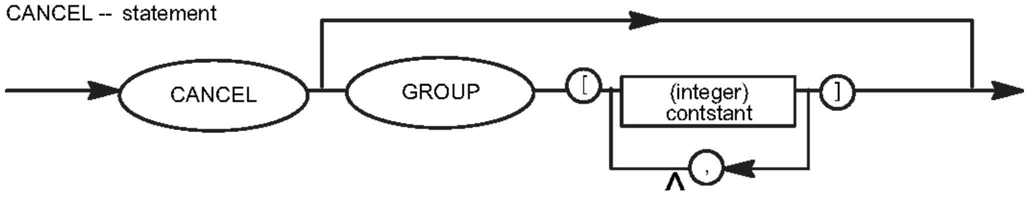
assignmemnt -- statement



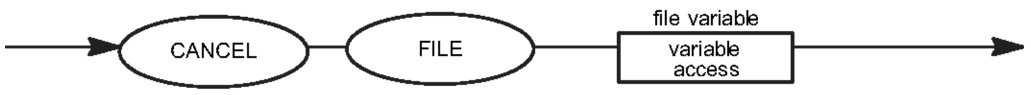
CALL -- routine



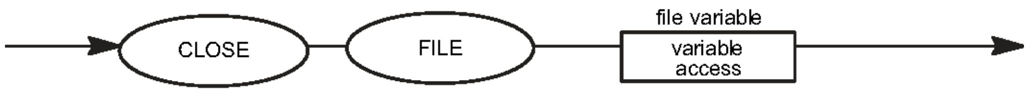
CANCEL -- statement



CANCELFILE -- statement



CLOSEFILE -- statement



CLOSEHAND -- statement

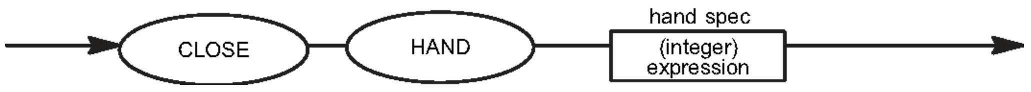
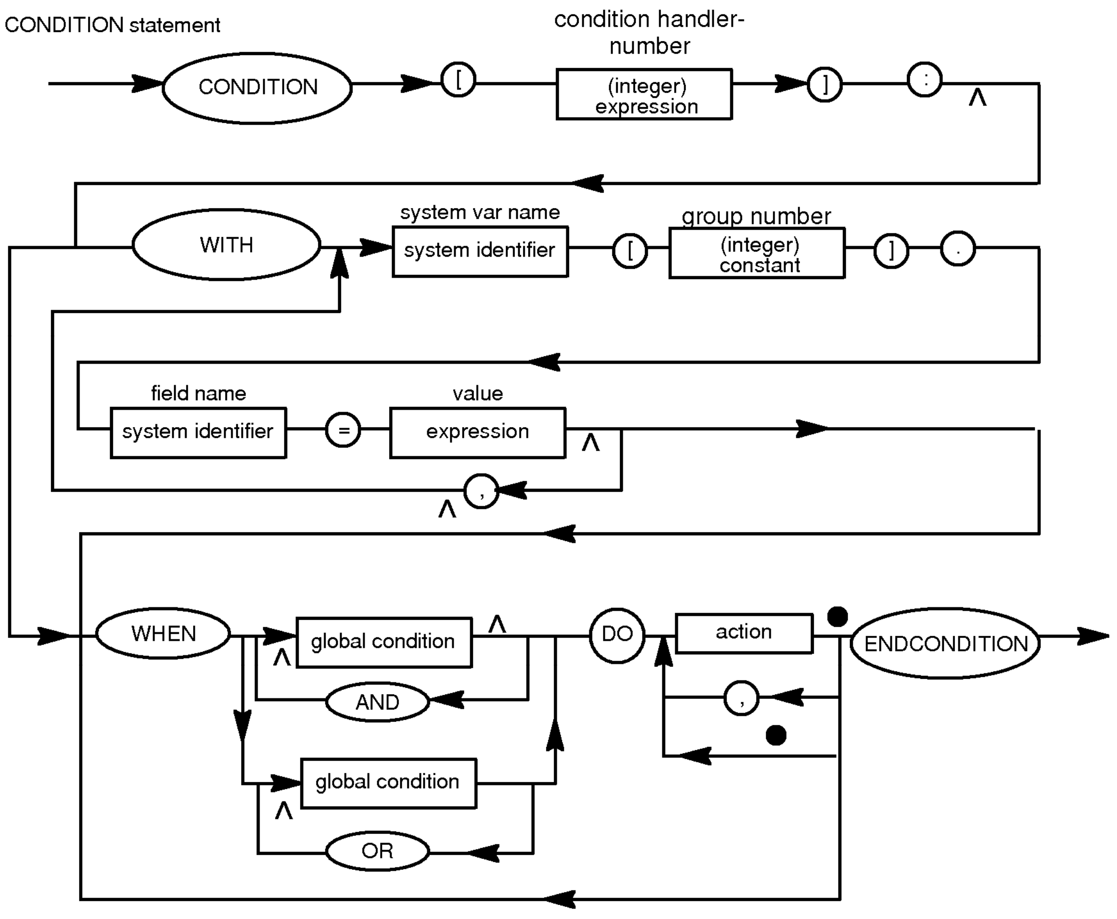
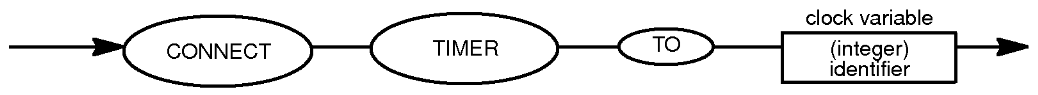


Fig. E(m)

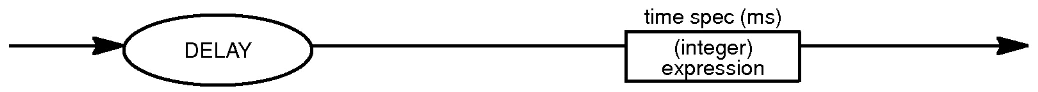
CONDITION statement



CONNECT TIMER — statement



DELAY — statement



DISABLE — statement

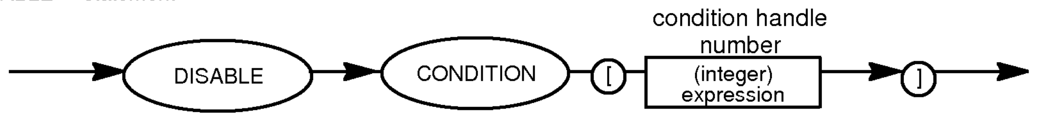
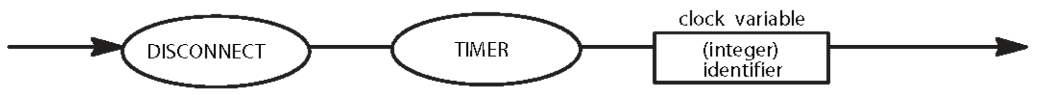
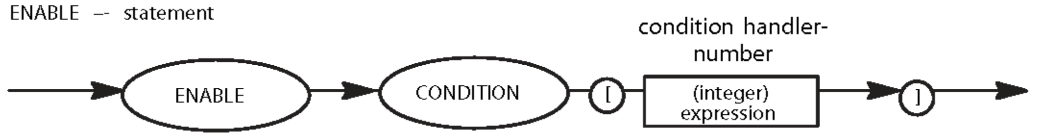


Fig. E(n)

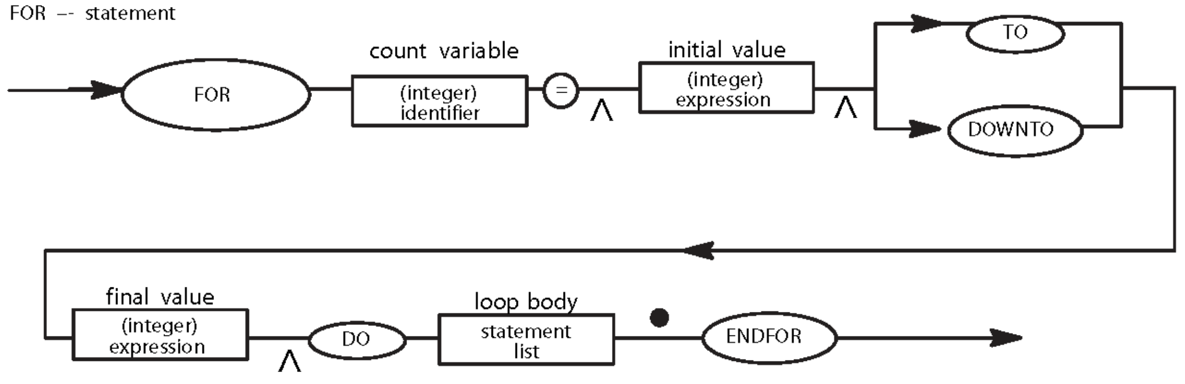
DISCONNECT TIMER -- statement



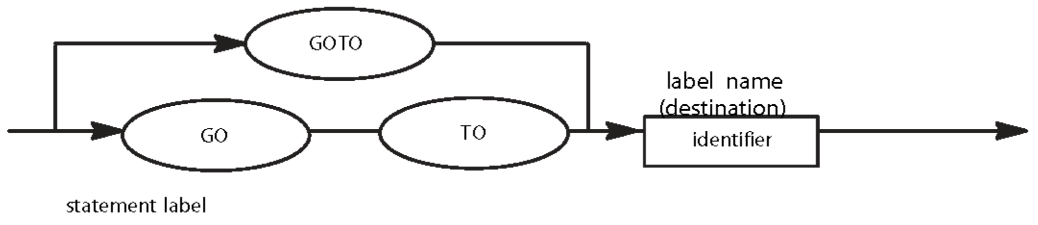
ENABLE -- statement



FOR -- statement



GO TO -- statement



IF THEN -- statement

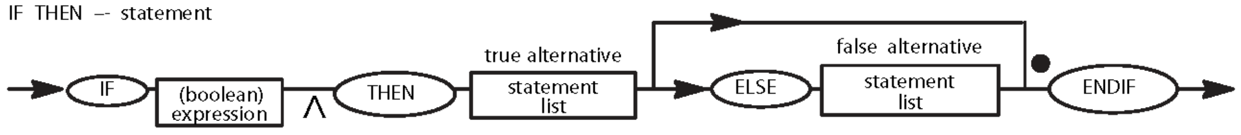
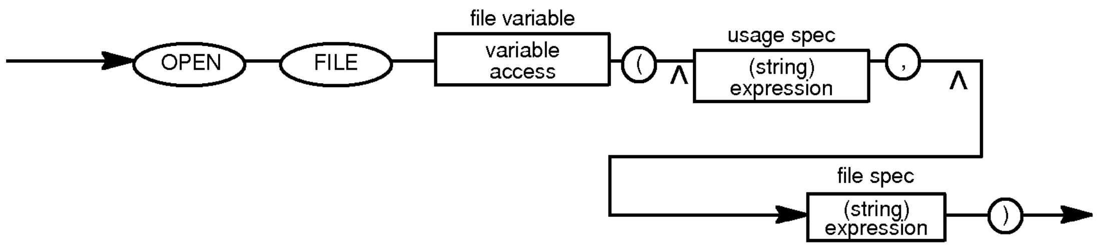
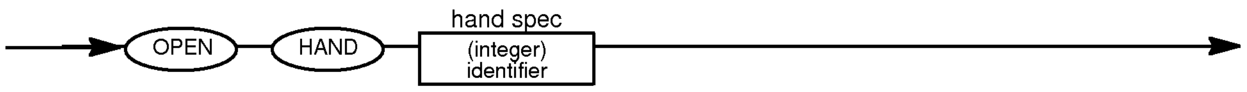


Fig. E(o)

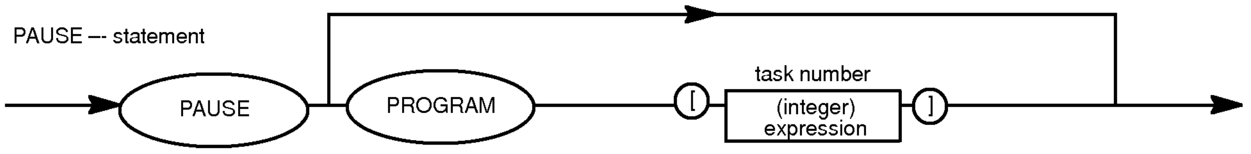
OPEN FILE –- statement



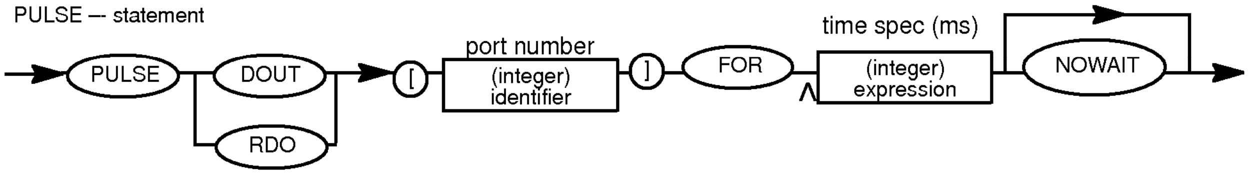
OPEN HAND — statement



PAUSE –- statement



PULSE — statement



PURGE — statement

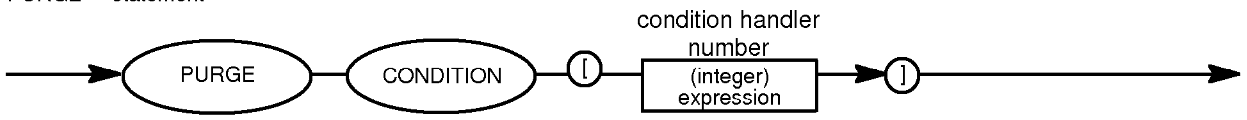
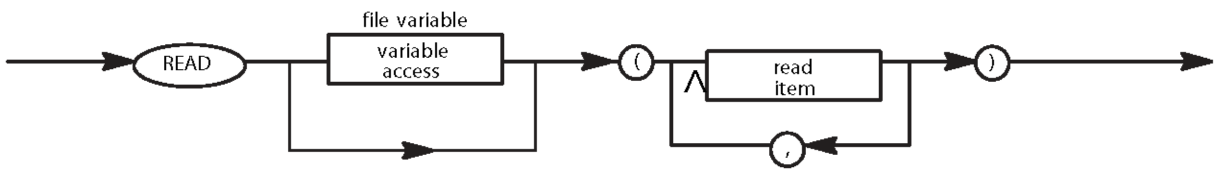
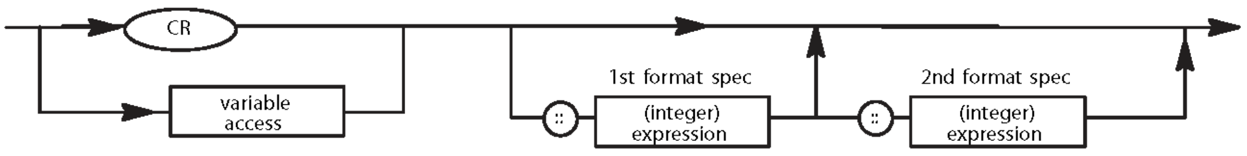


Fig. E(p)

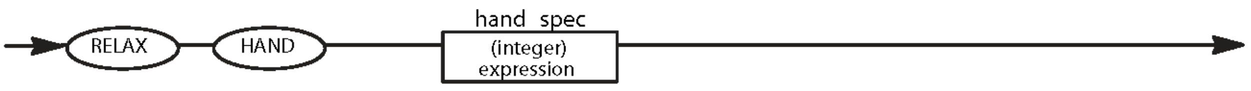
READ –- statement



read item –- statement



RELAX –- statement



REPEAT –- statement

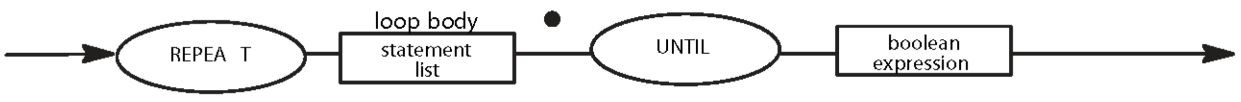
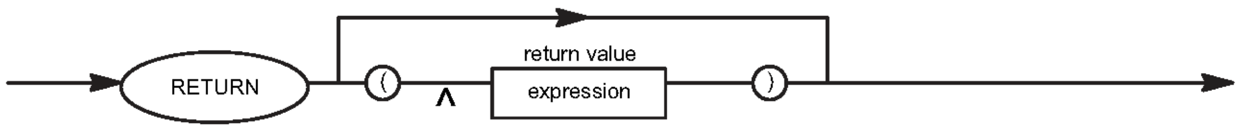
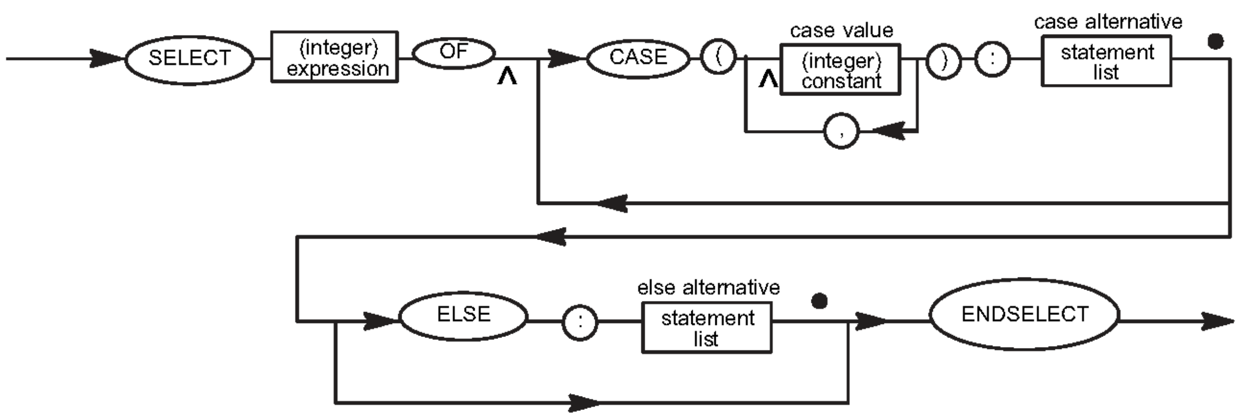


Fig. E(q)

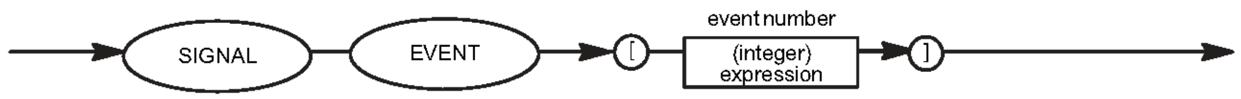
RETURN-- statement



SELECT-- statement



SIGNAL-- statement



USING-- statement

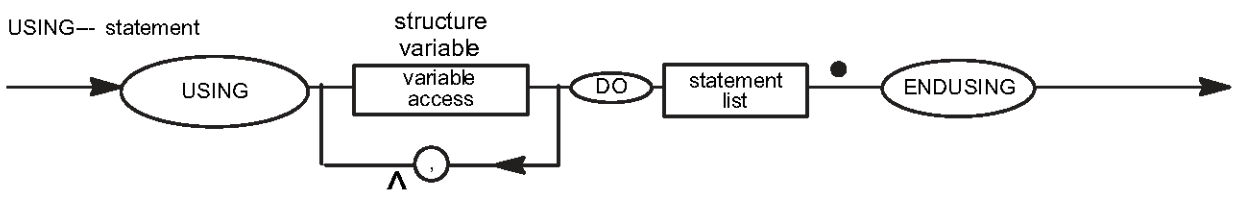
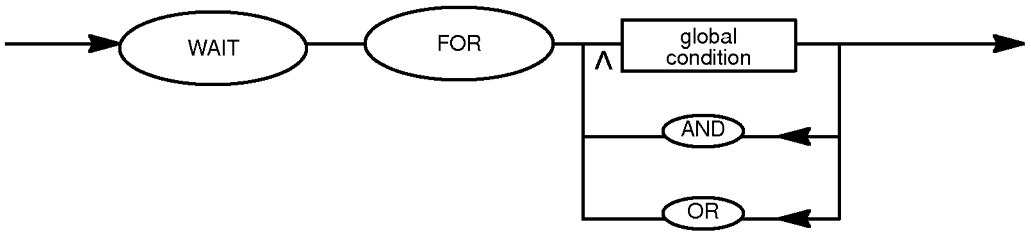
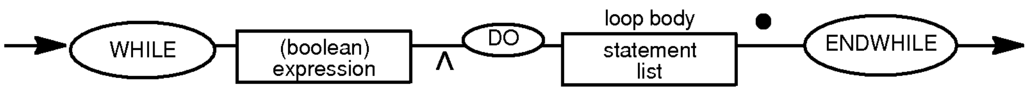


Fig. E(r)

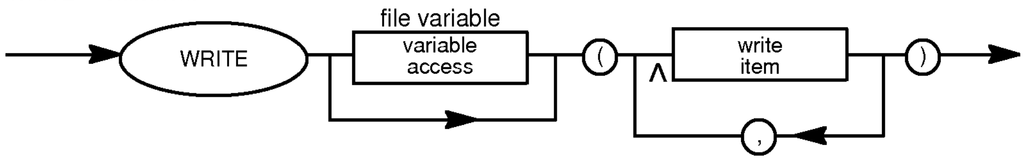
WAIT — statement



WHILE — statement



WRITE — statement



write item

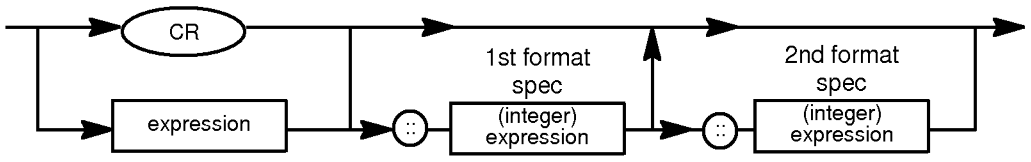


Fig. E(s)

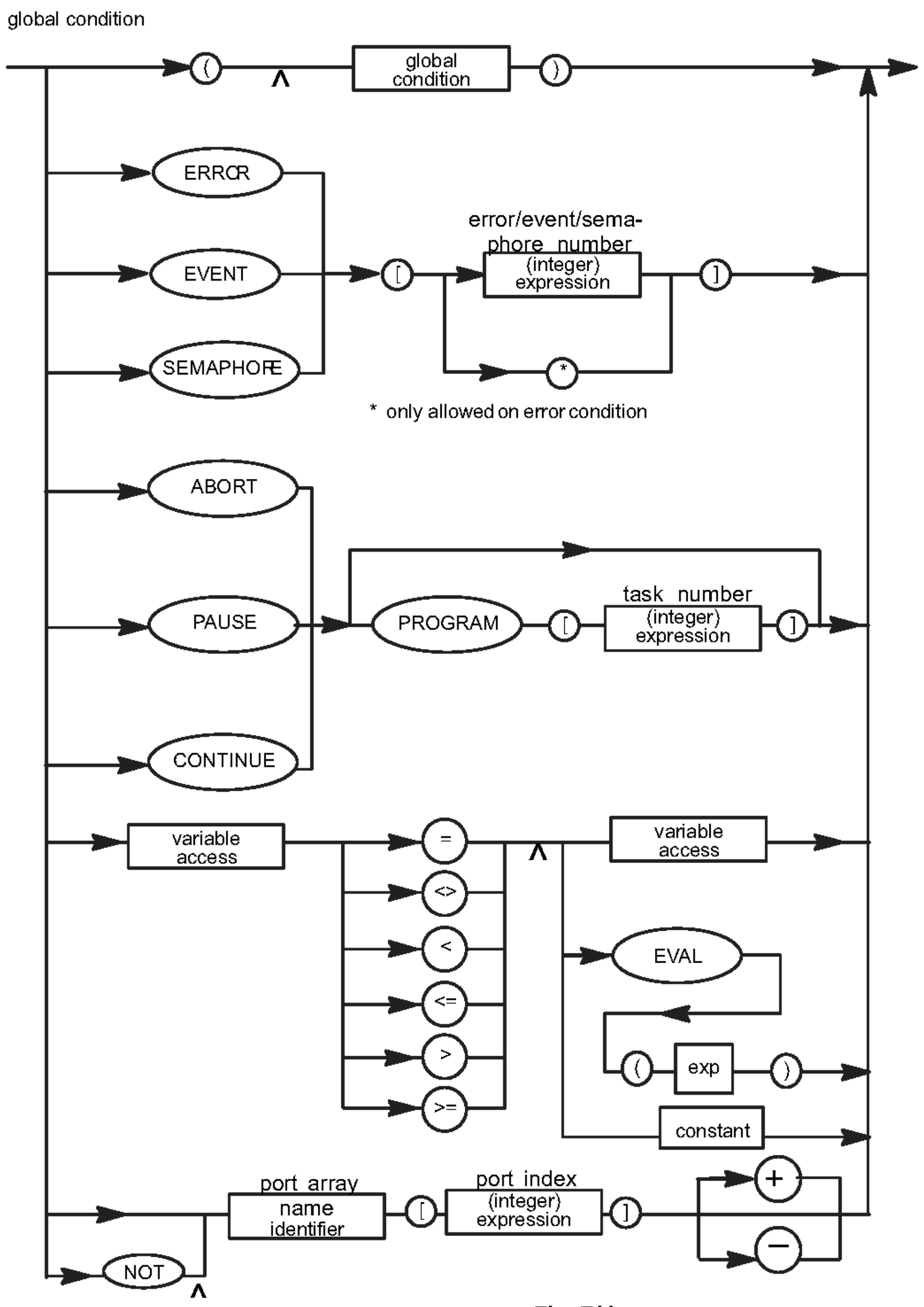

global condition
(
^
global condition
)
ERROR
EVENT
SEMAPHORE
error/event/sema-phore number
(integer) expression
[
]
*
* only allowed on error condition
ABORT
PAUSE
CONTINUE
PROGRAM
task number
(integer) expression
variable access
=
<>
<
<=
>
>=
EVAL
exp
constant
port array name identifier
port index (integer) expression
+
−
NOT

Fig. E(t)

condition handler action

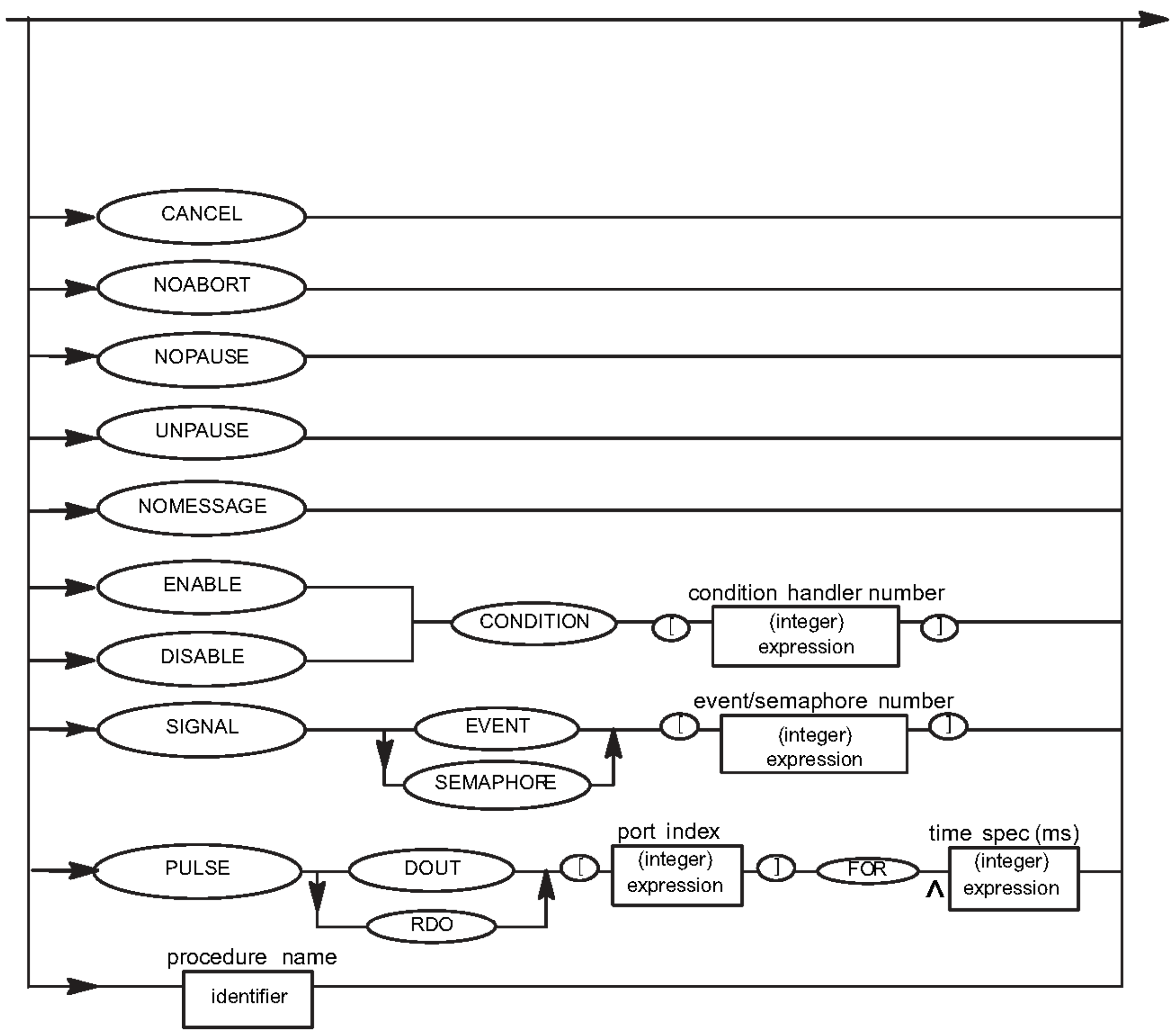


Fig. E(u)

condition handler action continued

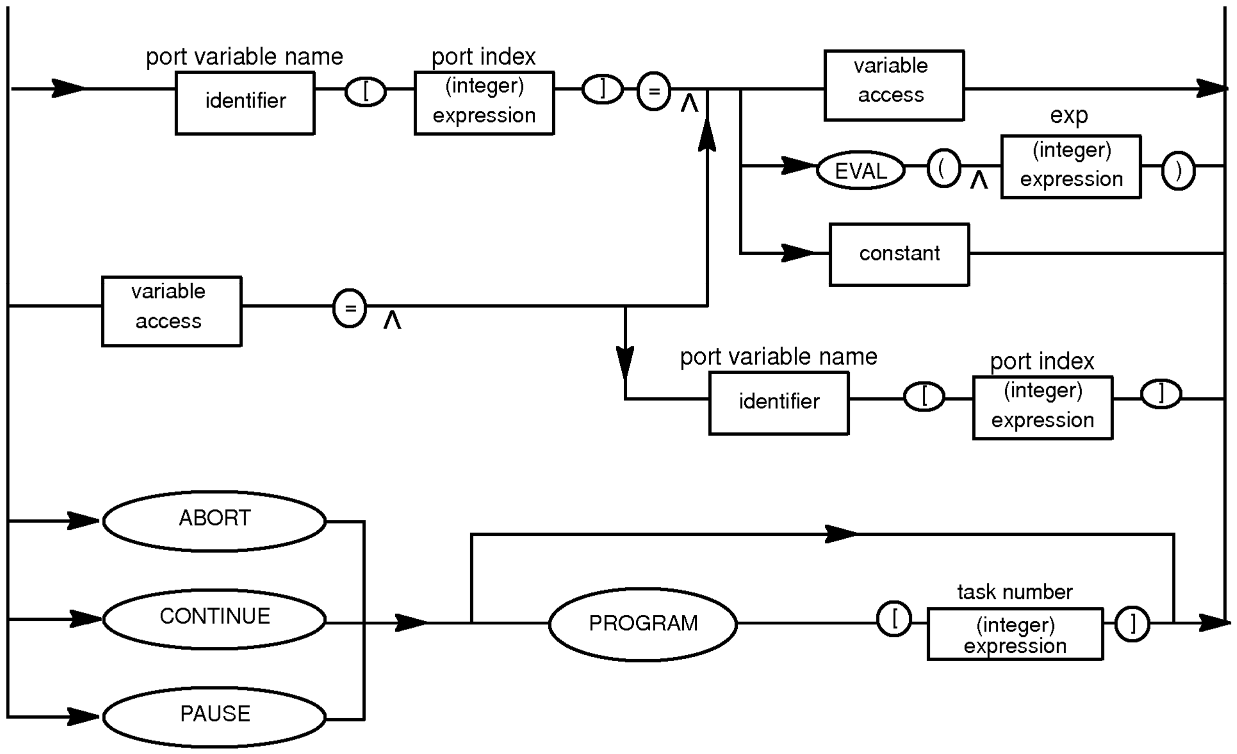
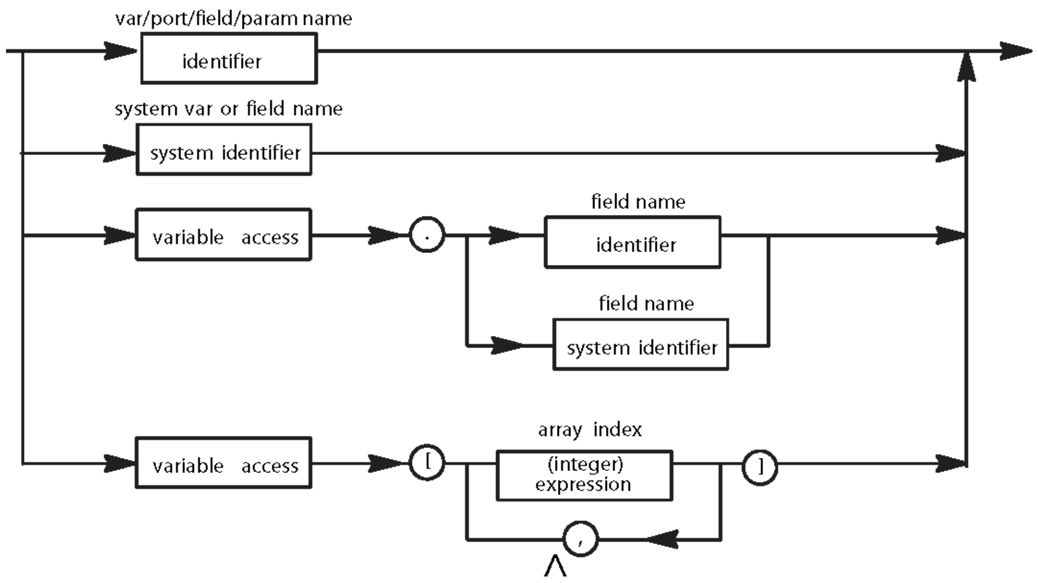


Fig. E(v)

variable access



expression

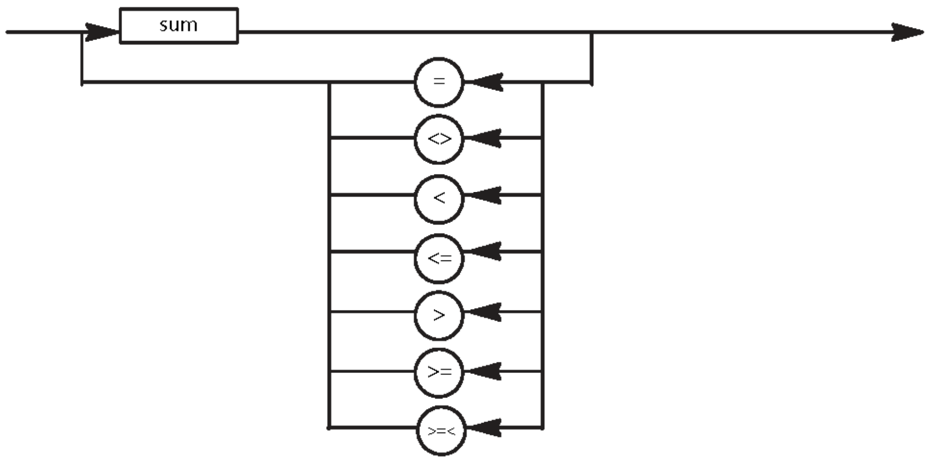
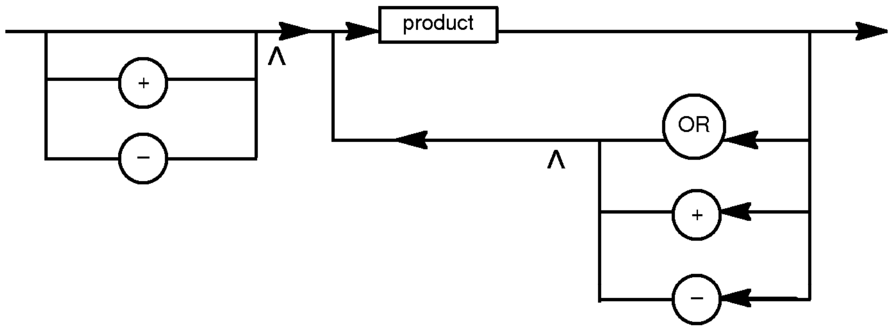
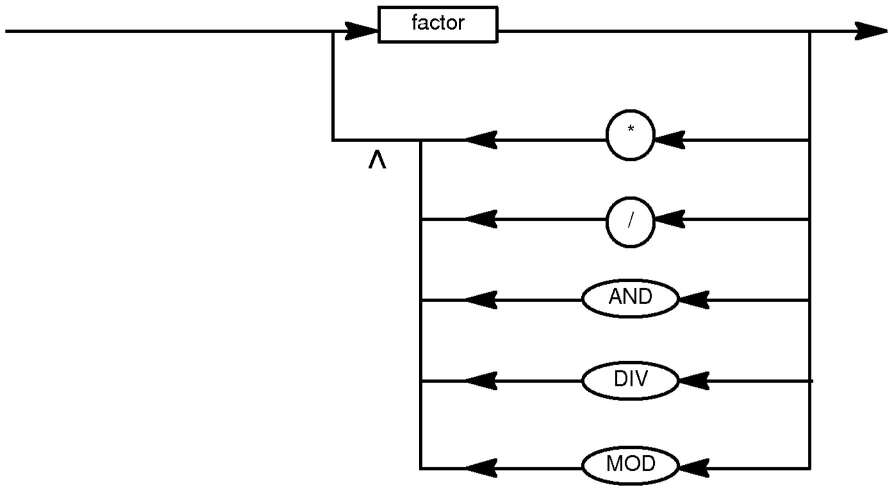


Fig. E(w)

sum



product



factor

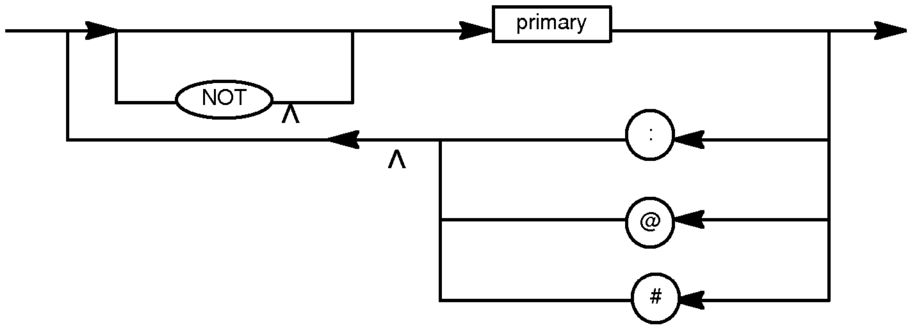
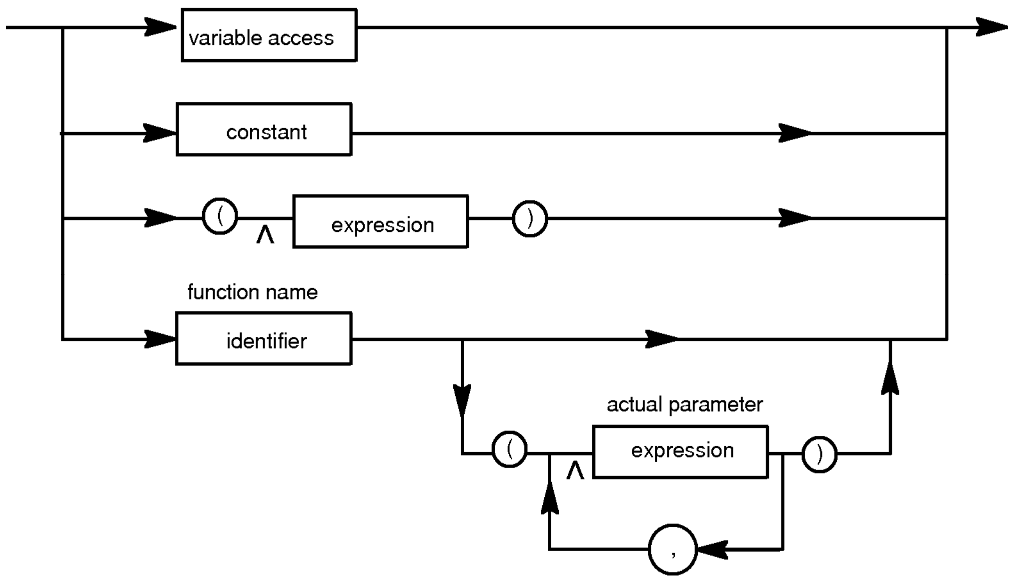


Fig. E(x)

primary



literal

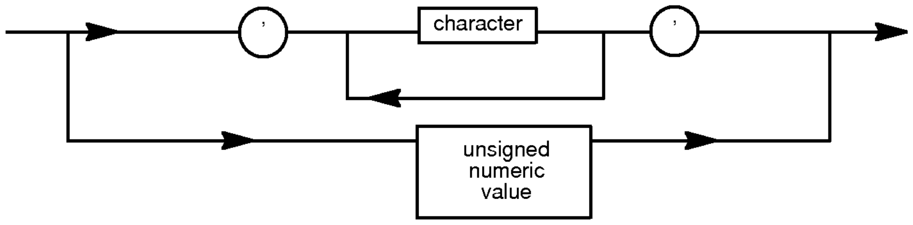


Fig. E(y)

F TRANSLATING BY ROBOGUIDE

This Chapter shows how to create or translate KAREL programs by ROBOGUIDE. Refer to ROBOGUIDE help to know the detail way to use ROBOGUIDE.

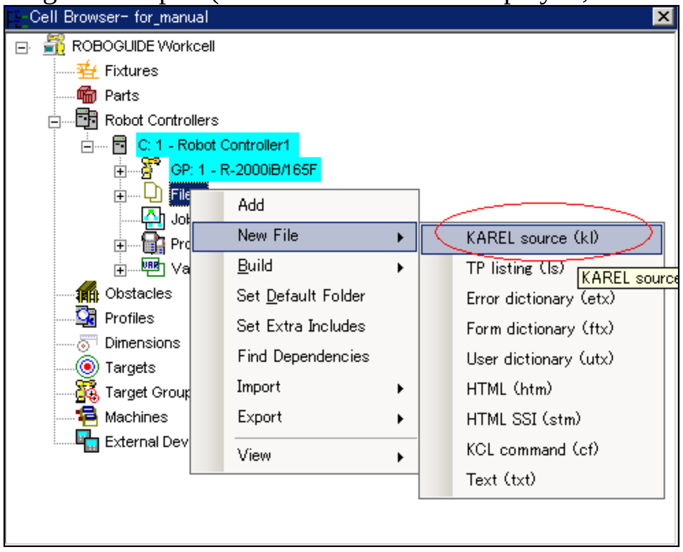
Start ROBOGUIDE and newly create or start the virtual robot.

F.1 TRANSLATE KAREL PROGRAMS

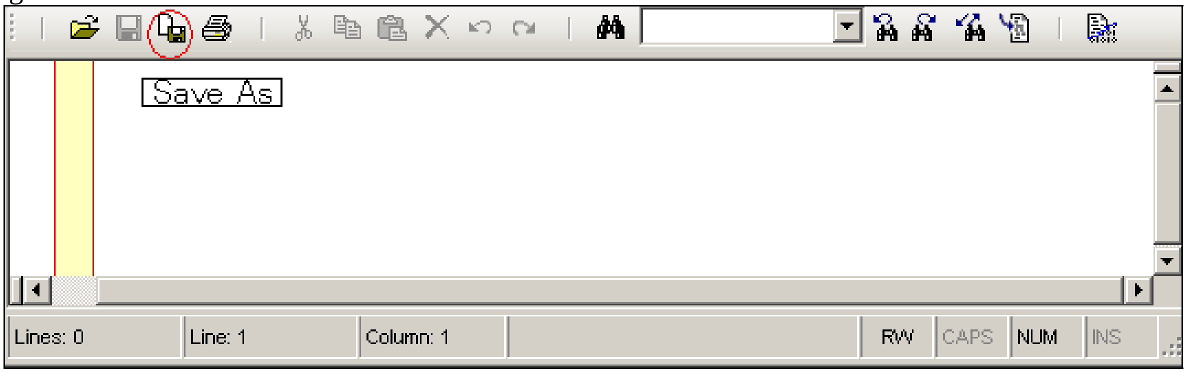
This section shows the sample of translating KAREL programs which is newly created or added.

F.1.1 The Sample of Newly Adding a KAREL Program

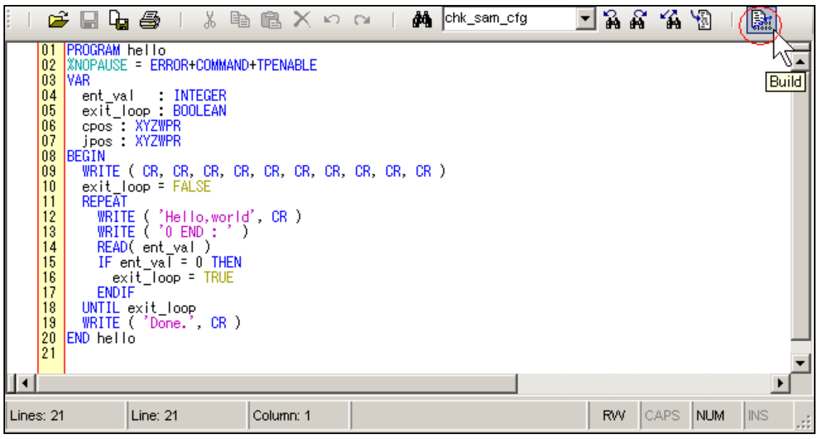
Right-click on cell browser, select new file and "KAREL source". Select "Add" in case of adding created KAREL program. Following is a sample. (If cell browser is not displayed, select from menu.)



A newly added file is empty. It is named like "Untitled1.kl". Click "Save As" button and input KAREL program name within 8 characters such as formtest.kl.

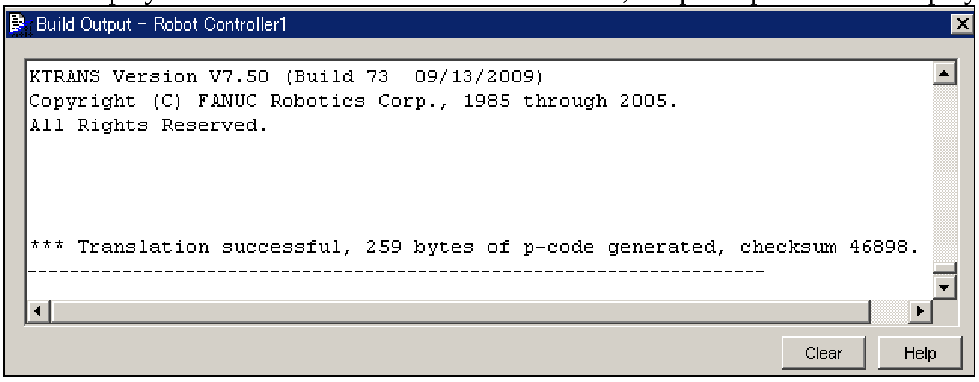


After inputting the KAREL program, click upper right build button, then translating the KAREL program. Translated KAREL program is loaded to virtual robot automatically if it is successfully built.



```
PROGRAM hello
%NOPAUSE = ERROR+COMMAND+TPENABLE
VAR
  ent_val   : INTEGER
  exit_loop : BOOLEAN
  cpos : XYZWPR
  jpos : XYZWPR
BEGIN
  WRITE ( CR, CR, CR, CR, CR, CR, CR, CR, CR, CR, CR )
  exit_loop = FALSE
  REPEAT
    WRITE ( 'Hello,world', CR )
    WRITE ( '0 END : ' )
    READ( ent_val )
    IF ent_val = 0 THEN
      exit_loop = TRUE
    ENDIF
  UNTIL exit_loop
  WRITE ( 'Done.', CR )
END hello
```

Result of build is displayed on other window. If errors are found, simple explanation is displayed.

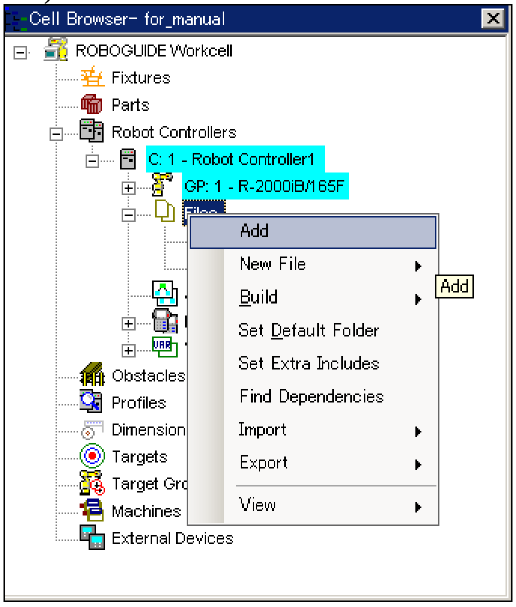


Build Output - Robot Controller1

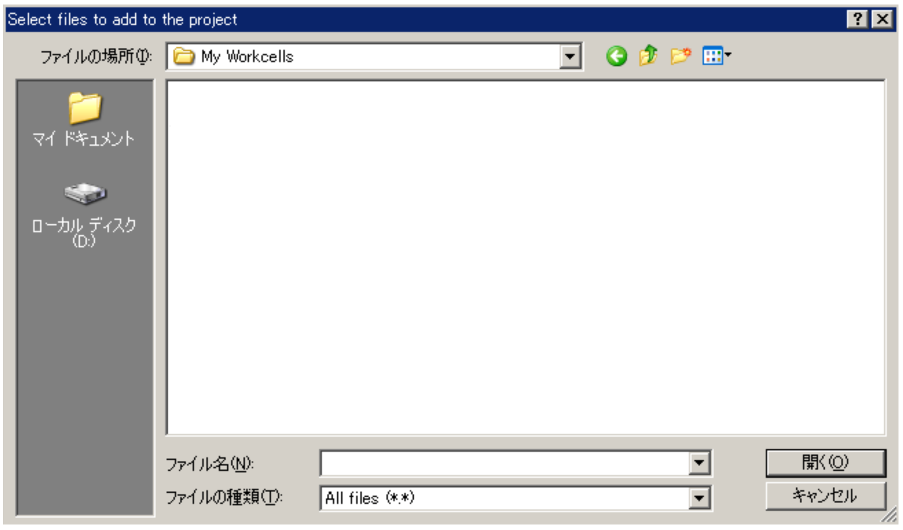
```
KTRANS Version V7.50 (Build 73  09/13/2009)
Copyright (C) FANUC Robotics Corp., 1985 through 2005.
All Rights Reserved.



*** Translation successful, 259 bytes of p-code generated, checksum 46898.
-------------------------------------------------------------------
```

F.1.2 The Sample of Adding Existed KAREL Program

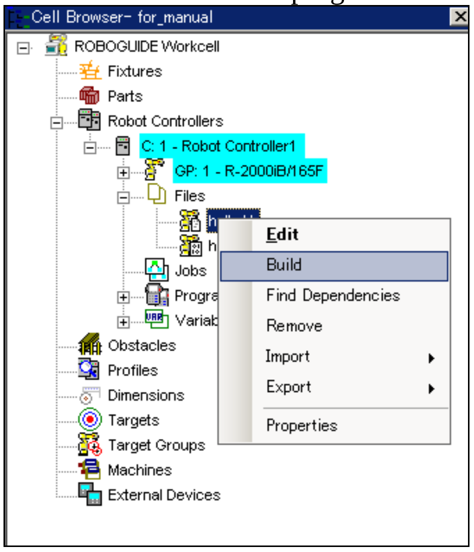
Right-click on file of cell browser and select “Add”. Following is the displaying sample. (If cell browser is not displayed, select from menu.)



Following screen is displayed, then select KAREL program to add.



Right-click on added file and select build. Then KAREL program is translated. You can translate dictionary files in a similar way to translation of KAREL programs.



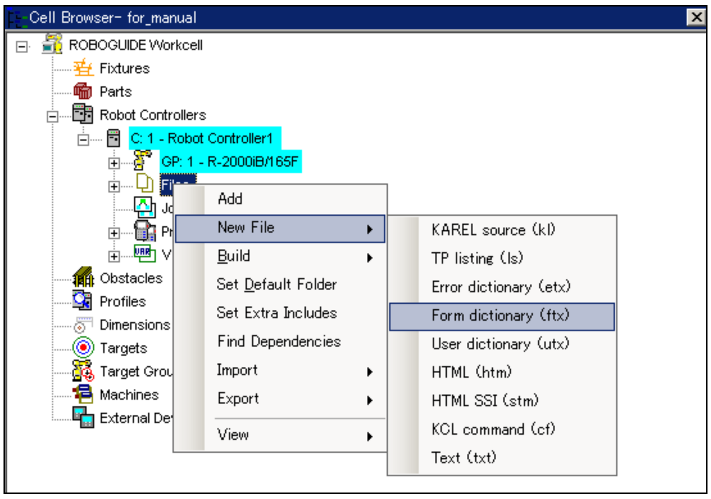
Like “The Sample of newly adding a KAREL program”, KAREL program can be built

F.2 TRANSELATE DICTIONARY

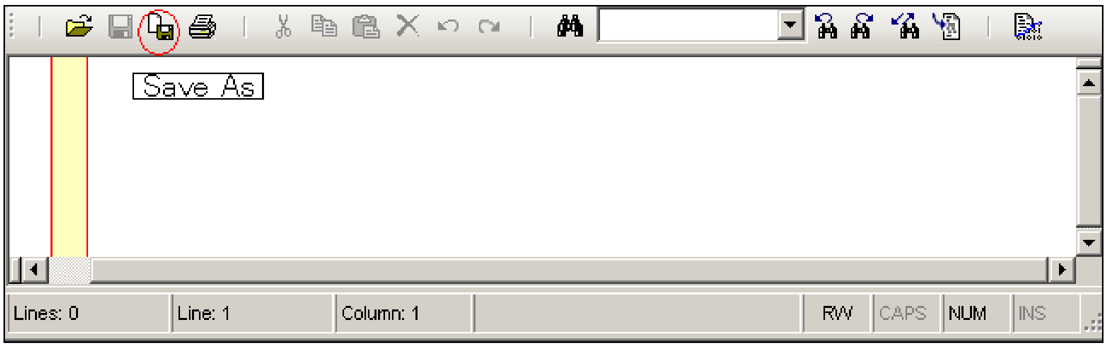
You can translate dictionary files in a similar way to translation of KAREL programs. Refer to ROBOGUIDE help to know creating dictionaries by ROBOGUIDE.

F.2.1 The Sample of Translating Dictionaries newly created

Right-click on cell browser, select “new file”. Following is the sample. (If cell browser is not displayed, select from menu.)

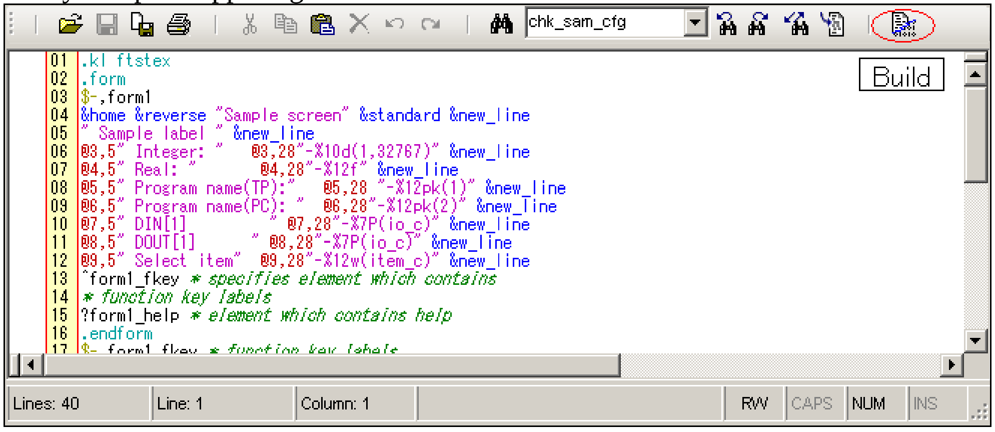


A newly added file is empty. It is named like “Untitled1.ftx”. Click “Save As” button and input dictionary name according to rule that first 4 characters are dictionary name and next 4 characters are language name. For example, about “dictengl.ftx”, “dict” is dictionary name and “engl” is language name.



Three files, flstengl.ftx, flstjapa.ftx, flstkanj.ftx, are prepared here for robot of multi languages. Each dictionary is English, Japanese (KANA: for legacy pendant) and Japanese (KANJI: for *i*Pendant).
ftstengl.ftx (for English)

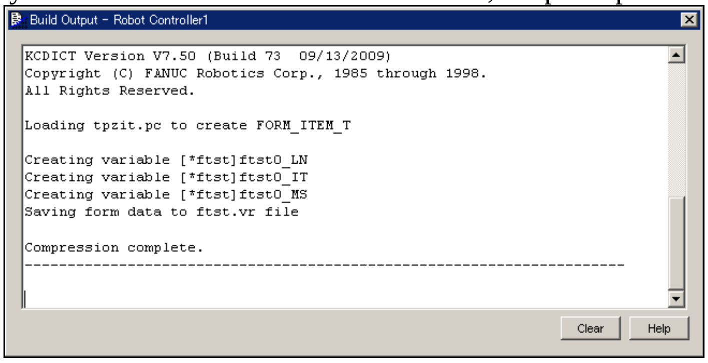
Create dictionary and push upper right build button.



```
.kl ftstex
.form
$-,form1
&home &reverse "Sample screen" &standard &new_line
" Sample label " &new_line
@3,5" Integer: "   @3,28"-%10d(1,32767)" &new_line
@4,5" Real: "       @4,28"-%12f" &new_line
@5,5" Program name(TP):"   @5,28 "-%12pk(1)" &new_line
@6,5" Program name(PC): "  @6,28"-%12pk(2)" &new_line
@7,5" DIN[1]           " @7,28"-%7P(io_c)" &new_line
@8,5" DOUT[1]        " @8,28"-%7P(io_c)" &new_line
@9,5" Select item"  @9,28"-%12w(item_c)" &new_line
^form1_fkey * specifies element which contains
* function key labels
?form1_help * element which contains help
.endform
$- form1_fkey * function key labels
```

Lines: 40 Line: 1 Column: 1 RW CAPS NUM INS

Result of build is displayed on other window. If errors are found, simple explanation is displayed.

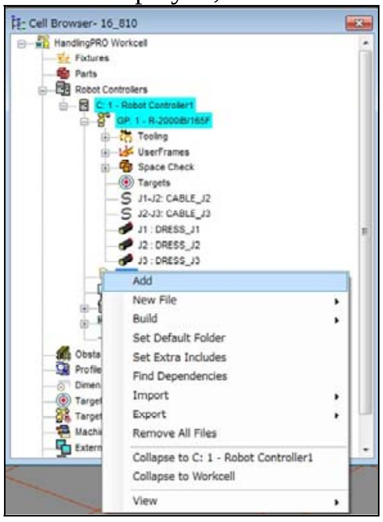


```
KCDICT Version V7.50 (Build 73  09/13/2009)
Copyright (C) FANUC Robotics Corp., 1985 through 1998.
All Rights Reserved.

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.
------------------------------------------------------------------------
```

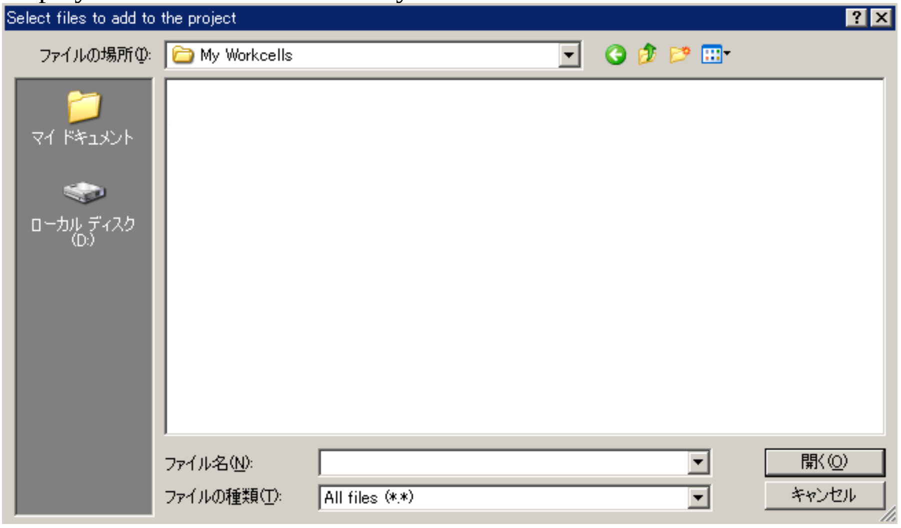
If dictionary is successfully translated, it is automatically loaded to virtual robot.

F.2.2 The Sample of Translating Dictionaries Added

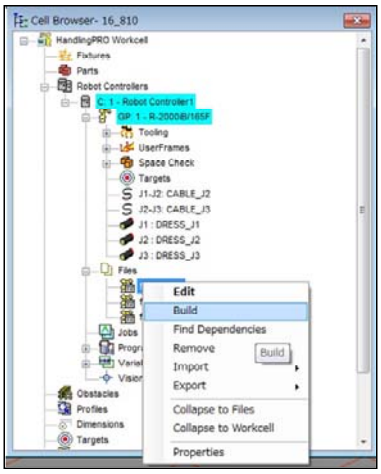
Select "Add" in the case of adding dictionary that is already created. Right-click on the file of cell browser and select "Add".
Following is a sample. (If cell browser is not displayed, select from menu.)



Select menu is displayed and select the dictionary to add.



Right-click on the added dictionary file on cell browser and select “Build”, and then it is built.



F.2.3 Dictionary Name and Languages

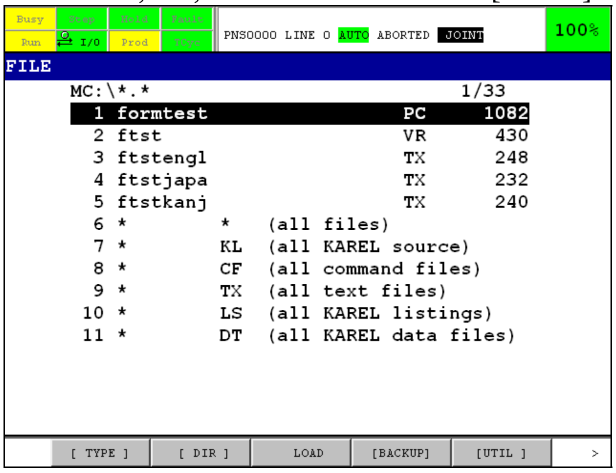
Dictionary name has following rules.

- File name is 8 characters.
- First 4 characters are dictionary name.
- Last 4 characters are language name.
- The extension is one of etx, ftx or utx.
- Ftx is to use Form Editor.
- Following is language name of 4 characters.
  "ENGL" = ENGLISH
  "JAPA" = JAPANESE
  "KANJ" = KANJI
  "FREN" = FRENCH
  "GERM" = GERMAN
  "SPAN" = SPANISH
  "CHIN" = CHINESE
  "TAIW" = TAIWANESE

F.3 LOAD TO THE ROBOT CONTROLLER

After confirming on virtual robot, next is to confirm on actual robot.

1. Copy PC, VR, TX files to Compact FLASH ATA card (called MC: hereinafter).
2. Insert the MC: to the robot controller.
3. Display FILE screen. It allows PC, VR, TX files to load with F3[LOAD]


```
Busy  Step  Hold  Fault
Run   I/O   Prod  TCyc     PNS0000 LINE 0 AUTO ABORTED JOINT    100%
FILE
      MC:\*.*                                          1/33
        1 formtest                            PC       1082
        2 ftst                                VR        430
        3 ftstengl                            TX        248
        4 ftstjapa                            TX        232
        5 ftstkanj                            TX        240
        6 *            *    (all files)
        7 *           KL    (all KAREL source)
        8 *           CF    (all command files)
        9 *           TX    (all text files)
       10 *           LS    (all KAREL listings)
       11 *           DT    (all KAREL data files)

      [ TYPE ]  [ DIR ]   LOAD   [BACKUP]  [UTIL ]   >
```

NOTE

Only FTST.VR is used because VR file includes starting position and width of data. VR file is created each language but enabled data is only one of them. Then you must unify kind, position, width of data in each language. Only one VR file is loaded. Usually, we recommend English VR file to use.

INDEX

REVISION RECORD

Edition	Date	Contents
01	Sep.,2012	

B-83144EN-1/01

* B - 8 3 1 4 4 E N - 1 / 0 1 *