FANUC Robot series

R-30*i*B/R-30*i*B Mate/R-30*i*B Plus/R-30*i*B Mate Plus/
R-30*i*B Compact Plus/R-30*i*B Mini Plus CONTROLLER

External Vision Interface

OPERATOR'S MANUAL

B-84244EN/02

ORIGINAL INSTRUCTIONS

Thank you very much for purchasing a FANUC robot.

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

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

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

SAFETY PRECAUTIONS

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

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

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

PERSONNEL

Personnel can be classified as follows:

Operator:

- Turns the robot controller power on/off
- Starts the robot program from the operator panel

Programmer or Teaching operator:

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

Maintenance technician:

- Operates the robot
- Teaches the robot inside the safeguarded space
- Performs maintenance (repair, adjustment, replacement)

The operator is not allowed to work in the safeguarded space.

The programmer or teaching operator and maintenance technician are allowed to work in the safeguarded space. Work carried out in the safeguarded space include transportation, installation, teaching, adjustment, and maintenance.

To work inside the safeguarded space, the person must be trained on proper robot operation.

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

Table s-1 Work Performed Outside the Safeguarded Space

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

Start robot program with teach pendant		○	○
Reset alarm with operator's panel		○	○
Reset alarm with teach pendant		○	○
Set data on teach pendant		○	○
Teaching with teach pendant		○	○
Emergency stop with operator's panel	○	○	○
Emergency stop with teach pendant	○	○	○
Operator Panel maintenance			○
Teach Pendant maintenance			○

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

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

DEFINITION OF SAFETY NOTATIONS

To ensure the safety of workers and prevent damage to the machine, this manual indicates each precaution on safety according to its severity. Read the contents of each **Warning** and **Caution** before attempting to use the robot.

⚠WARNING

Indicates a hazard resulting in the death or serious injury of the user could occur if he or she fails to follow the approved procedure.

⚠CAUTION

Indicates a hazard resulting in the injury of the user or damage to equipment could occur if the user fails to follow the approved procedure.

Note

Indicates a supplementary explanation not related to any WARNING or CAUTION.

TABLE OF CONTENTS

PART IV USING ROBOGUIDE TO TEST EVI INTERFACE

APPENDIX

PART I
OVERVIEW

Topics:

1 OVERVIEW

This manual explains how to properly set up and use the External Vision Interface option (Ext Mach Vis I/F Opt R897). There are example TP programs in Part II, section 3 which have examples of each routine call.

Note

For External Vision Interface, the stacksize of the TP Program which calls the EVI routines should be increased to 1000. If the user sees **INTP-302** stack overflow error they may need to increase the stack size of the TP program beyond 1000. If the problem persists, contact FANUC.

1.1 DESCRIPTION OF KAREL PROGRAMS

Table 1.1 Description of KAREL programs

Function	Description
MVS_CONNECT	Establish a connection to the machine vision system (MVS)
MVS_DISCO	Disconnect from the MVS
MVS_CALIB	Command the MVS to execute a calibration function and send optional position information to the mvs
MVS_SET_PROJECT	Set the active project for the MVS
MVS_LOCATE	Command the MVS to execute a process and store the results
MVS_GET_LOC	Retrieve one result which was received by the mvs_locate function, store it in registers accessible by the TP program
MVS_READ_BARCODE	Command the MVS to execute a barcode read, store resulting string in a numeric register
MVS_INSPECT	Command the MVS to execute a pass/fail inspection routine
MVS_SEND_CMD	Send a command not covered by the other functions, store the result in a string register accessible to the TP program.
MVS_SEND_SR	Use a string register (SR[]) to send a command not covered by the other functions, store the result in a string register accessible to the TP program.
MVS_PARSE	Pass in a delineated string, parse the first value and store it in another string register. This is to be used in conjunction with MVS_SEND_CMD.

PART II
FOR USERS

Topics:

1 SETUP AND CONFIGURATION

1.1 SETTING THE ROBOT IP ADDRESS

About this task

If you are using a virtual robot in ROBOGUIDE, skip this section when configuring the communications.

Note

Refer to Part IV, Chapter 1, USING ROBOGUIDE TO TEST THE EVI INTERFACE for how to set up a virtual robot in ROBOGUIDE.

Before you begin

Prior to using the external vision interface, you must configure communication. The machine vision system (MVS) and the robot controller must be connected via Ethernet.

Procedure

1. Press **MENU**.
2. Select **Setup**.
3. Select **Host Comm**.
4. Select **TCP/IP** and set up the IP address of the robot controller. Make sure to use the > key and select **F3, INIT**, to initialize this setting once configured. You will see a screen similar to the following.

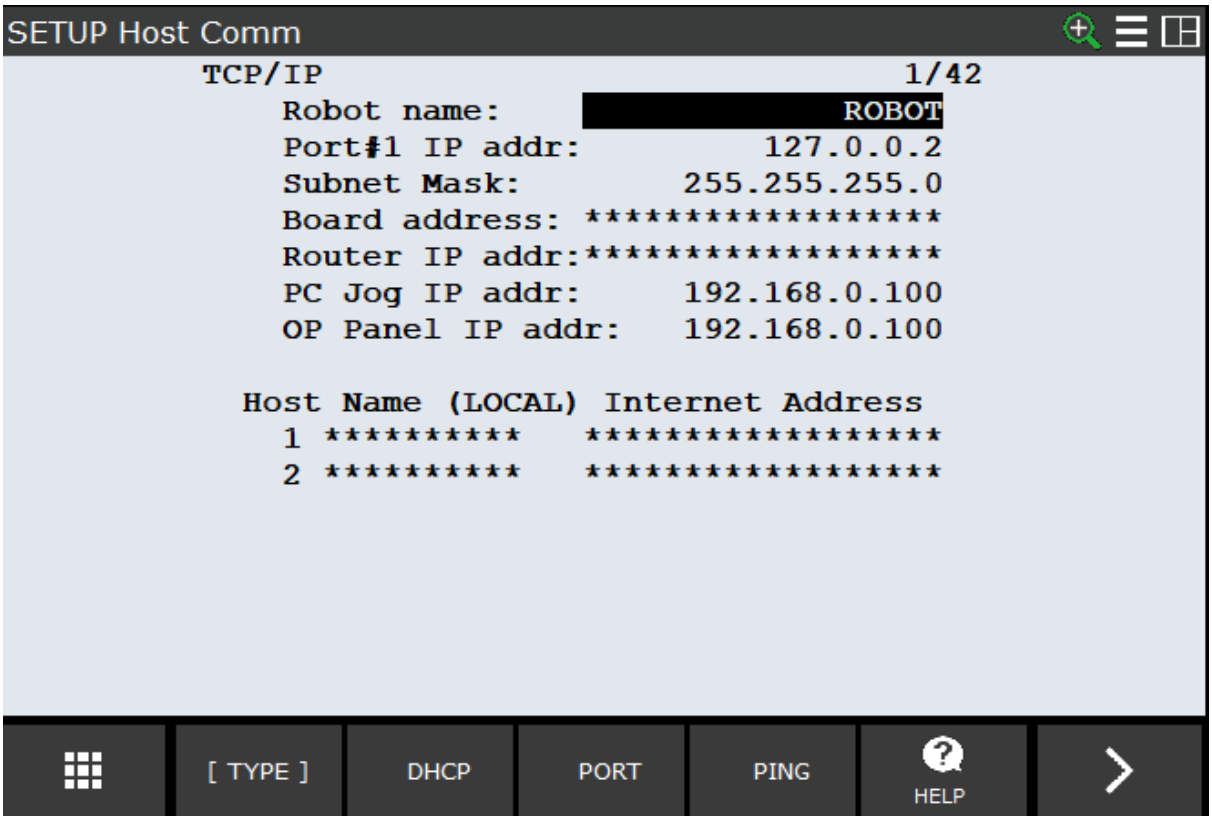


Figure 1.1 TCP/IP Address Setup

What to do next

Create a client tag for the communication with the MVS.

1.2 SETTING UP A CLIENT TAG

Procedure

1. Press **MENU**.
2. Select **Setup**.
3. Select **Host Comm**.
4. Press **F4, [SHOW]**
5. Select **Clients**
6. Select one of the 8 tags to set up.
7. Set the protocol to SM (socket messaging), the Server IP/Hostname should be the IP address of the external Machine Vision System.
8. Set up the port number to communicate to the vision system. Vision systems should have a port number defined in their documentation.

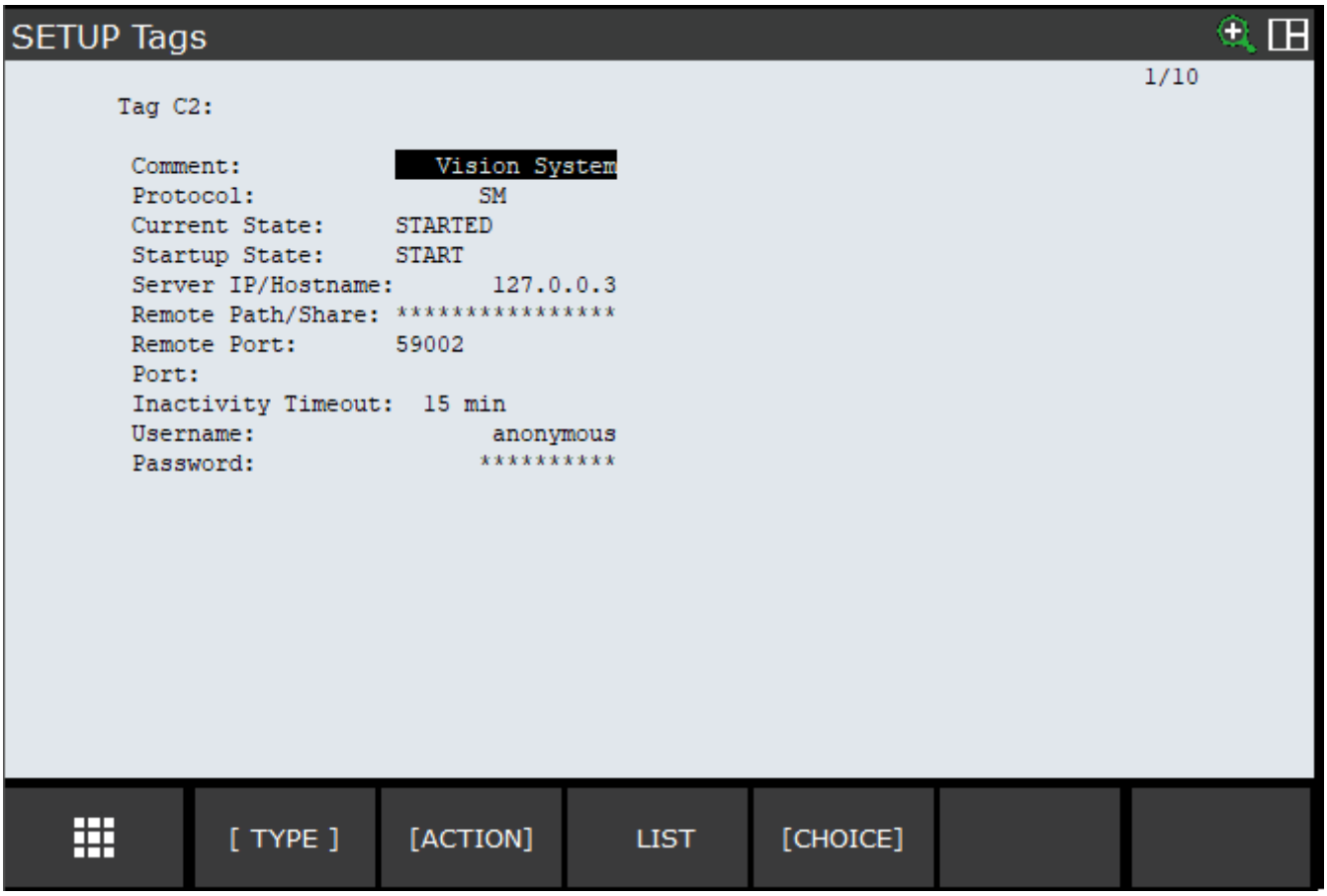


Figure 1.2 (a) Client Tag Setup Screen

9. Once this is set up, select **F2, ACTION** and select **DEFINE**, then select **F2, ACTION** and select **START**.

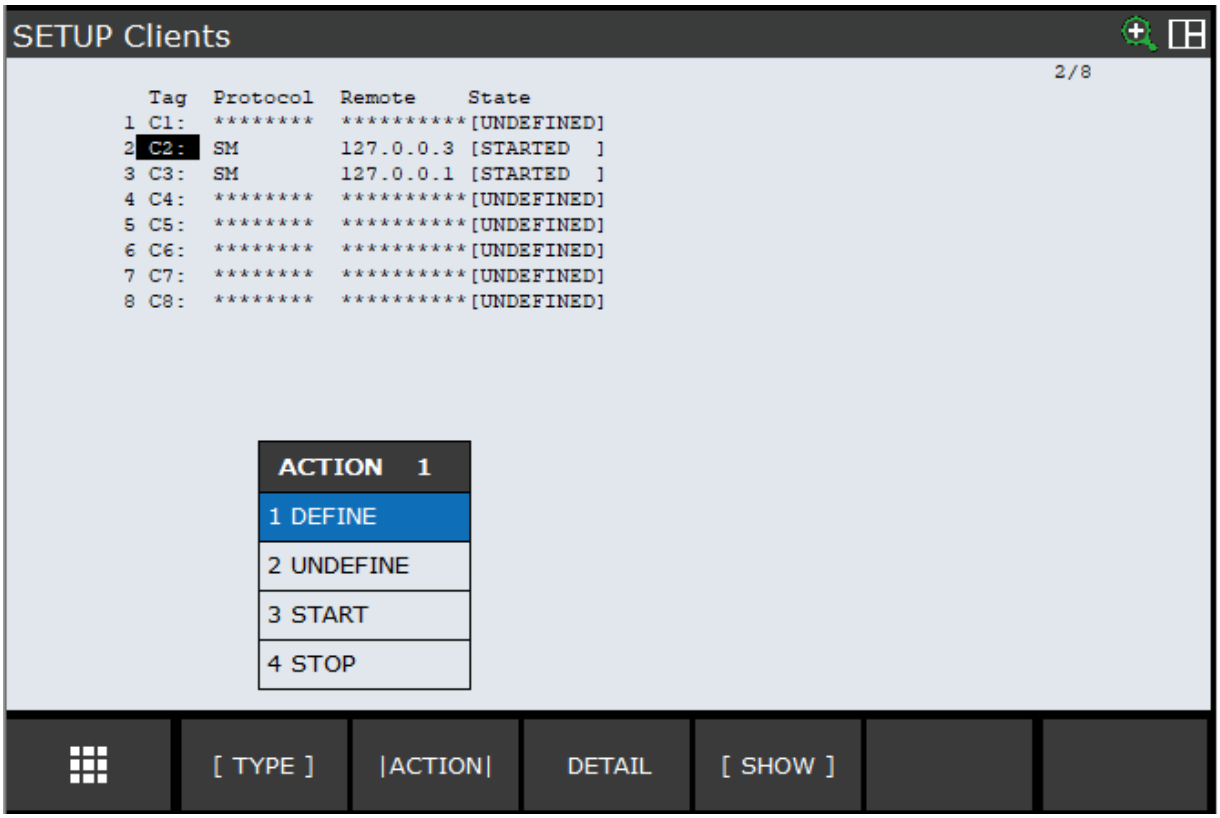


Figure 1.2 (b) Client Tag Define/Start Actions

This option supports connections for any client tag (1-8), and currently supports up to two simultaneous active connections to systems. Programs may be run asynchronously, but the status value should be checked in order to make sure that data is not acted upon prior to the MVS sending it.

2 KAREL PROGRAMS

There are several status values which the MVS programs use internally (meaning these statuses are from the programs themselves, not statuses returned by the MVS.) These are listed below:

Table 2 Status Values

Value	Description
-1	Waiting for response from MVS
0	Success
1	MVS sent “NOTREADY” in MVS_STATUS
1	MVS sent “FAILURE” in MVS_LOCATE command
3	The message the MVS sent is invalid, it doesn't match up with the data format expected.
11	Timeout occurred, the MVS took too long to respond to the request or there was no response
12	Not Connected. The socket messaging connection was not established.

2.1 MVS_CONNECT

`MVS_CONNECT` establishes communication to an MVS system. The timeout value (in seconds) will be used for all KAREL program calls to the connected MVS until the connection is disconnected. This connection will stay active until `MVS_DISCO` is called.

Example of an `MVS_CONNECT` call from a TP program:

```
CALL MVS_CONNECT("Server ID"=2,"Status R[]"=3,"Timeout (sec)"=10) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful.

[in] Timeout (sec): INTEGER - The number of seconds the program will attempt to establish the connection to the MVS before posting an error and aborting the connection attempt.

2.2 MVS_DISCO

This program is used to disconnect the connection to the MVS. The status will return 0 when successful.

```
CALL MVS_DISCO("Server ID"=2,"Status R[]"=65) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[out] Status R[]: INTEGER - Numeric register index to use for status. It is populated with a non-zero error status if there is an error disconnecting, or a 0 if successful

2.3 MVS_CALIB

MVS_CALIB is used to send data to calibrate a camera system. This program sends position data to the MVS if the robot group number and PR[] to send are specified. If either the robot group or PR[] to send arguments are equal to 0, no position data will be sent.

```
CALL MVS_CALIB("Server ID"=2,"Process Name"='START_CALIB.33',"Status
R[]"=85,"Robot group"=1,"PR[] to send"=0) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Process Name : STRING - The job ID to send to the MVS

[out] Status R[] : INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful.

[in] Robot group : INTEGER - Robot group is the group number of the robot whose position data you wish to send (if needed). Set this argument to 0 to send no position data, or set PR[] to send to zero.

[in] PR[] to send: INTEGER - If the calibration process requires position data, store it in a PR[] and pass the PR[] index into this argument, along with the group number of the robot whose position needs to be sent.

2.4 MVS_SET_PROJECT

Some vision systems require loading/setting the project prior to executing a vision task. This function facilitates that purpose.

```
CALL MVS_SET_PROJECT("Server ID"=2,"Project Name"='Project
1',"Status R[]"=86) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Project Name: STRING - refers to the project you wish to command the MVS to set as the active project.

[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful.

2.5 MVS_LOCATE

This is the main function to call when you want the vision system to perform a vision process. The parameters provided allow a vision system to respond with multiple results (up to 100) in one call, as well as up to 10 measurements per result.

```
CALL MVS_LOCATE("Server ID"=2,"Process Name"='VISPROC1',"Status
R[]"=101,"numfound R[]"=102,"Error SR[]"=3,"Robot Grp num"=0,"PR[]
to send"=0) ;
```

[in] Server ID : INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Process Name: STRING - The job ID to send to the MVS

[out] Status R[] : INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful

[out] Numfound R[]: INTEGER - Register index for the number of found results. If no results are found, this will be 0. Whether or not the MVS returns success or failure in this situation is dependent on the MVS.

[out] Error SR[]: INTEGER - String Register index to populate with error text, if any is returned from the MVS. Some vision systems may just return a numeric value with no accompanying text. Please refer to that vision system's documentation for more information.

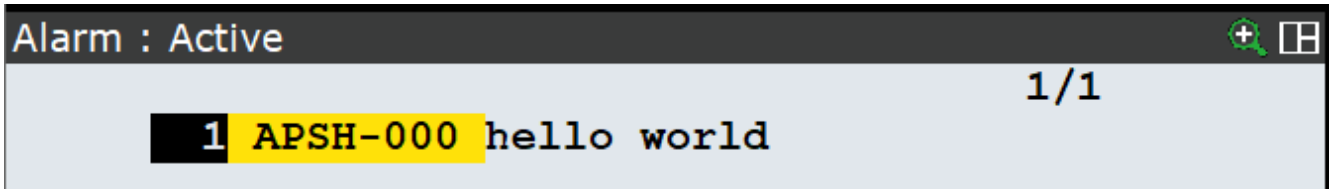
Note

If 0 or a negative value is passed for the Error SR[] argument, the error text from the MVS will not be stored in a string register.

[in] Robot grp num: INTEGER - Robot group number associated with the position data in the position register referenced by PR[] to send (if desired). Pass 0 in if you do not wish to send position data.

[in] PR[] to send: INTEGER - Index for position data to be sent (if desired). Pass 0 in for no position data to be sent.

If an error is returned, the KAREL program will post the error text provided by the MVS as a string in the alarm log with a generic error facility code. There is such an example below (with "hello world" as the error text). If the MVS does not return an additional string after the non-zero status, the following message will be posted as an alarm instead `status from MVS = %d, check MVS's manual` where %d is replaced by the vision status value.



Note

The final data block which is sent must terminate in CRLF (CHR(13) + CHR(10)). This indicates there is no more data coming for this return string.

2.6 MVS_GET_LOC

This function is used to retrieve buffered data which was received by the `MVS_LOCATE` program. It must be used in conjunction with the `MVS_LOCATE` program in order for data to be available in position and numeric registers, even if only a single result was returned by `MVS_LOCATE`.

```
CALL MVS_GET_LOC("Server ID"=2,"Process Name"='VISPROC1',"Result to
get"=1,"Status R[]"=104,"Model ID R[]"=105,"Result PR[]"=50,"Num
Meas R[]"=106,"Meas R[] start"=110) ;
```

[in] Server ID : INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Process Name: STRING - corresponds to jobID which was sent for the MVS_LOCATE command which you want to retrieve results data for

[in] Result to get: INTEGER - The desired result index to retrieve from the results (if only 1 result was returned then pass in a 1). In an iterative process for multiple results, this will likely be indirectly indexed with a register value in order to get the results sequentially.

[out] Status R[]: INTEGER – Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful.

[out] Model ID R[]: INTEGER – Numeric register index for model ID if the same vision process is used to identify multiple different types of parts. This lets the TP program know which type of part was found. If the process is only configured to identify a single object this value will be 1.

[out] Result PR[]: INTEGER – Position register index to populate with the returned offset or found position data from the MVS. It is the responsibility of the programmer to use the vision data returned from the MVS appropriately.

[out] Num Meas R[]: INTEGER – Numeric register index for number of measurements which are returned for this vision process

[in] Meas R[] Start: INTEGER – Starting numeric register index for measurements, the first measurement will be in this register, and subsequent registers will contain the remaining measurement values.

This function does not communicate directly to the MVS. It is used so the TP program can access the data returned from the MVS_LOCATE command.

If status > 0 : failure, no measurement data is stored to registers, and no position register data is modified. If the vision process returns success and returns 0 measurements (R[Arg7]=0), no measurement registers are populated.

Note

If 0 or a negative value is passed for Arg5, the model ID from the MVS will not be stored in a register.

Note

If 0 or a negative value is passed for Arg7 and/or Arg8, the measurement values from the MVS will not be stored in registers.

2.7 MVS_READ_BARCODE

This function is intended for use with a barcode read process and expects the MVS to return a string of up to 254 characters.

The status of the process will be stored in the Status R[] index, and the barcode string is stored in the SR[] return index. No robot position data is sent with this command.

```
CALL MVS_READ_BARCODE("Server ID"=2,"Process Name"='Barcode
A',"Status R[]"=87,"SR[] return"=4) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Process Name: STRING - The job ID to send to the MVS

[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful

[out] SR[] return: INTEGER- String register index to populate with the barcode string value returned by the MVS

2.8 MVS_INSPECT

This function is intended for use with pass/fail inspection processes. It should be used with processes which just return a true/false response. No robot position data is sent with this command.

```
CALL MVS_INSPECT("Server ID"=2,"Process Name"='Passfail 1',"Status
R[]"=84,"SR[] error"=5) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Process Name: STRING - The job ID to send to the MVS

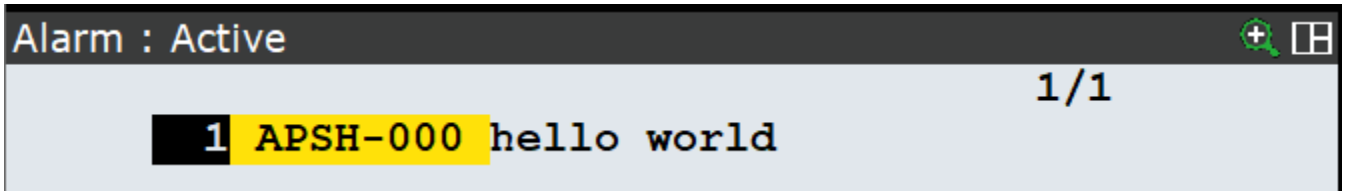
[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful

[out] SR[] error: INTEGER - String Register index to populate with error text if the MVS returned error text. Some vision systems may just return a numeric value with no accompanying text. Please refer to that vision system's documentation for more information.

Note

If 0 or a negative value is passed for the SR[] Error argument, the error text from the MVS will not be stored in a string register.

If an error is returned, the KAREL program will post the error text provided by the MVS as a string in the alarm log with a generic error facility code. There is such an example below (with "hello world" as the error text). If the MVS does not return an additional string after the non-zero status, the following message will be posted as an alarm instead `status from MVS = %d, check MVS's manual` where %d is replaced by the vision status value.



2.9 MVS_SEND_CMD

`MVS_SEND_CMD` is used for generic messaging which may not fit in with the other defined functions. Due to internal handling of Teach Pendant program strings, only 34 characters may be sent with this command. If you wish to send more than 34 characters, see `MVS_SEND_SR` which allows you to send a string register instead.

```
CALL MVS_SEND_CMD("Server ID"=2,"Command string"='This Is A Vision
Command,1,2,3',"Status R[]"=103,"SR[] for result"=6) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] Command string : STRING - This argument can contain any string you wish to send to the MVS, up to 34 characters long. This 34 character limit is due to internal handling of teach pendant program strings.

[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful

[out] SR[] for result: INTEGER - The string register index which is populated with the return string sent by the MVS.

2.10 MVS_SEND_SR

MVS_SEND_SR is used for generic messaging which may not fit in with the other defined functions. It sends a string to the MVS which is stored in a string register (SR[]), which can contain up to 254 characters.

```
CALL MVS_SEND_SR("Server ID"=2,"SR[] to send"=1,"Status
R[]"=103,"SR[] for result"=6) ;
```

[in] Server ID: INTEGER - corresponds to the client tag configured to communicate with the Machine Vision System (MVS)

[in] SR[] to send: INTEGER - This is the string register index whose contents will be sent to the MVS.

[out] Status R[]: INTEGER - Numeric register index to use for status. It is set to -1 while the program is processing the action. Once the operation is complete, the status register will be populated with either an internal status if there is a communication failure, the return status of the MVS if there was an error on the MVS side, or 0 to indicate no errors and the operation was successful

[out] SR[] for result: INTEGER - The string register index which is populated with the return string sent by the MVS.

2.11 MVS_PARSE

MVS_PARSE is used in tandem with MVS_SEND_CMD to parse out (separate) each element in the return string. The SR[] to parse string will be modified when this function runs, so it will contain the original string minus the first delimiter and the string before it. The delimiter is a ‘,’ character.

```
CALL MVS_PARSE("SR[] to parse"=6,"SR[] for result"=7) ;
```

[in] SR[] to parse: INTEGER - This is the SR[] index containing the entire string which is to be parsed.

[out] SR[] for result: INTEGER - This is the SR[] index which will contain the value parsed from the string.

3 EXAMPLE PROGRAMS

3.1 EXAMPLE TP PROGRAM FOR CALIBRATION

In this example program, the robot moves to 33 positions and sends position data to the MVS in order to calibrate the vision system. It also sends a tool frame to the MVS.

Note

In some instances, it is useful to call MVS_DISCO prior to calling MVS_CONNECT because if you call MVS_CONNECT but the connection is already established, MVS_CONNECT will return a non-zero status to indicate the connection is already established.

```
1: CALL MVS_DISCO("Server ID"=2,"Status R[]"=65) ;
2: CALL MVS_CONNECT("Server ID"=2,"Status R[]"=3,"Timeout
(sec)"=20) ;
3: CALL MVS_CALIB("Server ID"=2,"Process
Name"=Start_calib,33','Status R[]"=85,"Robot group"=1,"PR[] to
send"=0) ;
4: !Calib example ;
5: PR[55:CALIB POS]=P[1:BOTTOM RIGHT] ;
6: PR[56,1:CALIB POS]=0 ;
7: PR[56,2:CALIB POS]=0 ;
8: PR[56,3:CALIB POS]=0 ;
9: PR[56,4:CALIB POS]=0 ;
10: PR[56,5:CALIB POS]=0 ;
11: PR[56,6:CALIB POS]=0 ;
12: R[60:CALIB INDEX]=0 ;
13: ;
14: FOR R[61]=0 TO 2 ;
15: FOR R[62]=0 TO 2 ;
16: FOR R[63]=0 TO 2 ;
17: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
18: ;
19: R[64:X OFFSET]=R[63]*(-250) ;
20: R[65:Y OFFSET]=R[62]*250 ;
21: R[66:Z OFFSET]=R[61]*87.5 ;
22: ;
23: PR[56,1:CALIB POS]=R[64:X OFFSET] ;
24: PR[56,2:CALIB POS]=R[65:Y OFFSET] ;
25: PR[56,3:CALIB POS]=R[66:Z OFFSET] ;
26: ;
27:L PR[55:CALIB POS] 200mm/sec CR0 OFFSET, PR[56];
28: SR[5]=SR[4]+R[60:CALIB INDEX] ;
29: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
30: IF R[85]>0,JMP LBL[4] ;
31: ;
32: ENDFOR ;
33: ENDFOR ;
34: ENDFOR ;
35: ;
36: !ROTATION POSITIONS ;
```

```
37: R[60:CALIB INDEX]=28 ;
38: SR[5]=SR[4]+R[60:CALIB INDEX] ;
39: PR[55:CALIB POS]=P[10] ;
40:L PR[55:CALIB POS] 100mm/sec FINE ;
41: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
42: IF R[85]>0, JMP LBL[4] ;
43: ;
44: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
45: SR[5]=SR[4]+R[60:CALIB INDEX] ;
46: PR[55:CALIB POS]=P[11] ;
47:L PR[55:CALIB POS] 100mm/sec FINE ;
48: ;
49: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
50: IF R[85]>0, JMP LBL[4] ;
51: ;
52: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
53: SR[5]=SR[4]+R[60:CALIB INDEX] ;
54: PR[55:CALIB POS]=P[12] ;
55:L PR[55:CALIB POS] 100mm/sec FINE ;
56: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
57: IF R[85]>0, JMP LBL[4] ;
58: ;
59: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
60: SR[5]=SR[4]+R[60:CALIB INDEX] ;
61: PR[55:CALIB POS]=P[13] ;
62:L PR[55:CALIB POS] 100mm/sec FINE ;
63: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
64: IF R[85]>0, JMP LBL[4] ;
65: ;
66: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
67: SR[5]=SR[4]+R[60:CALIB INDEX] ;
68: PR[55:CALIB POS]=P[14] ;
69:L PR[55:CALIB POS] 100mm/sec FINE ;
70: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
71: IF R[85]>0, JMP LBL[4] ;
72: ;
73: R[60:CALIB INDEX]=R[60:CALIB INDEX]+1 ;
74: SR[5]=SR[4]+R[60:CALIB INDEX] ;
75: PR[55:CALIB POS]=P[15] ;
76:L PR[55:CALIB POS] 100mm/sec FINE ;
77: CALL MVS_CALIB("Server ID"=2,"Process Name"=SR[5],"Status
R[]"=85,"Robot group"=1,"PR[] to send"=55) ;
78: IF R[85]>0, JMP LBL[4] ;
79: ;
80: PR[4]=UTOOL[2:pointer] ;
81: CALL MVS_CALIB("Server ID"=2,"Process
Name"='SET_TOOL_FRAME',"Status R[]"=85,"Robot group"=1,"PR[] to
send"=4) ;
82: IF R[85]>0, JMP LBL[4] ;
83: LBL[4] ;
84: !ERROR HANDLING CODE WOULD GO HERE ;
```

3.2 EXAMPLE TP PROGRAM FOR BIN PICKING

In this example program, the robot uses the MVS to locate a bin of parts, then goes into a loop in which it tells the MVS to locate the next part to pick, the robot picks the part then drops the part off. When the MVS returns a non-zero status, the code would jump to a label which would handle the error based on the status code, which may include a status to say there are no parts left.

```
1: !BIN PICK EXAMPLE ;
2: CALL MVS_CONNECT("Server ID"=2,"Status R[]"=3,"Timeout
(sec)"=10) ;
3: CALL MVS_SET_PROJECT("Server ID"=2,"Project
Name"='LOAD_TASK.13',"Status R[]"=86) ;
4: CALL MVS_LOCATE("Server ID"=2,"Process
Name"='POSE_PALLET.13',"Status R[]"=101,"numfound R[]"=102,"Error
SR[]"=3,"Robot Grp num"=0,"PR[] to send"=0) ;
5: IF R[101]<>0,JMP LBL[999]
6: LBL[42] ;
7: CALL MVS_LOCATE("Server ID"=2,"Process
Name"='POSE_WORK.13',"Status R[]"=101,"numfound R[]"=102,"Error
SR[]"=3,"Robot Grp num"=0,"PR[] to send"=0) ;
8: IF R[101]<>0,JMP LBL[999]
9: CALL MVS_GET_LOC("Server ID"=2,"Process
Name"='POSE_WORK.13',"Result to get"=1,"Status R[]"=104,"Model ID
R[]"=105,"Result PR[]"=60,"Num Meas R[]"=106,"Meas R[] start"=110) ;
10: !IF ERROR STATUS ;
11: IF R[101:MVS STATUS]<>0,JMP LBL[999] ;
12: !MOVE TO APPROACH ;
13:L PR[50:PICK POS] 150mm/sec FINE Tool_Offset,PR[60] ;
14: !PICK THE PART ;
15:L PR[50:PICK POS] 100mm/sec FINE TB 0.20sec, CALL GRIP_PART ;
16: !RETREAT;
17:L PR[50:PICK POS] 150mm/sec FINE Tool_Offset,PR[60] ;
18: !DROP OFF PART ;
19:L PR[51:DROP POS] 100mm/sec FINE Tool_Offset,PR[60] ;
20:L PR[51:DROP POS] 100mm/sec FINE TA 0.10sec, CALL DROP_PART ;
21: !RETREAT;
22:L PR[51:DROP POS] 100mm/sec FINE Tool_Offset,PR[60] ;
23: !GET NEXT PART ;
24: JMP LBL[42] ;
25: LBL[999] ;
26: !ERROR HANDLING CODE WOULD GO HERE ;
```

PART III
FOR DEVELOPERS

Topics:

1 DEVELOPING AN INTERFACE

If you are developing an interface for your vision system to communicate with this interface, additional information is required. This part of the manual is to assist developers in writing software for their vision system. The basic requirements are that the vision system communicates with socket messaging over an Ethernet connection. This section describes each routine's command string (which is sent to the MVS) and the expected format of the return string (which is sent from the MVS to the robot controller).
`MVS_CONNECT`, `MVS_DISCO`, and `MVS_GET_LOC` do not require specific code from the vision system, so they are not included in this section.

1.1 MVS_LOCATE

This KAREL program will send a command to run a vision task on the external MVS.

Refer to Part II, Section 2.5 for an explanation of arguments

1.1.1 MVS_LOCATE Example Command and Return Strings

The FANUC robot will send a command **LOCATE** followed by a string (max length 32 characters) denoting the MVS job to run and 6 real values (relative to the active tool frame and user frame) in XYZ(mm)WPR(degrees) Cartesian format. These 6 values will come from a PR[] if a PR index specified in the function call is a non-zero positive value.

Example string from FANUC 1:

```
LOCATE,BOX31416,1025.235,-500.236,400.002,12.25,-55.2,-63.35
```

LOCATE tells the MVS what type of command is being requested.

BOX31416 is the vision process the MVS should run.

The next 6 values are the x, y, z, w, p, r position data from a position register.

Example string from FANUC 2:

```
LOCATE,BOX31416
```

This is an example which does not send the position information (Arg6 and/or Arg7 = 0).

The MVS will respond with `SUCCESS` (or `0`) or `FAILURE` (or a non-zero integer).

For SUCCESS (or 0), the following data will follow:

- Number of found results (max of 100) Number of extra measurement values associated with each result (Max 10, only real values)

  Model ID (integer), this is to accommodate when a single vision process can identify multiple types of parts. Each unique type of part should have its own modelID for the vision process. In cases where a vision system only identifies one type of part, the modelID is still a required return value.

  XYZ(mm)WPR(degrees) found results

  Extra measurement data (real)

To send a negative value, we add a "-" at the top of string.

Example MVS_LOCATE return string 1:

```
SUCCESS,2,5,1,0.123,4.567,8.910,0.0,0.0,0.0,1.1,1.2,1.3,1.4,1.5,2,11.1
213,14.1516,17.1819,0,0,0,1.2,1.4,4.3,7.4,5.5
```

Example MVS_LOCATE return string 2:

```
0,2,5,1,0.123,4.567,8.910,0.0,0.0,0.0,1.1,1.2,1.3,1.4,1.5,2,11.1213,14
.1516,17.1819,0,0,0,1.2,1.4,4.3,7.4,5.5
```

Explanation of example return string values:

- SUCCESS (or 0) is the vision status
- The number of found results is 2
- There are 5 measurement values associated with each result
- The first result's modelID is 1
- The next 6 values are the position of the first result (x=0.123, y=4.567, z=8.910, w=0.0, p=0.0, r=0.0 in this example)
- The following 5 values are measurements associated with the first result. measurement 1 = 1.1, measurement 2 = 1.2, measurement 3 = 1.3, measurement 4 = 1.4, measurement 5 = 1.5 in this example.
- The proceeding data is the Model ID, then position values, and then 5 measurement values for the second result.

For FAILURE a string of up to 246 characters may follow providing a reason for failure.

Example FAILURE return string:

```
FAILURE,GenericVision-9999 Parameter is invalid
```

```
2,GenericVision-9999 Parameter is invalid
```

```
2
```

If no string is provided after the error status, an error will be posted to the alarm log which states `status from MVS = %d, check MVS's manual`, where %d is replaced by the status value returned by the MVS.

1.2 MVS_CALIB

This KAREL program will tell the MVS to run a calibration process, or continue the next step in a calibration process.

Refer to Part II, Section 2.3 for an explanation of arguments.

1.2.1 MVS_CALIB Example Send and Return Strings

Example string from FANUC:

```
CALIB,viscal1,1.23,4.56,7.89,90,-120,40
```

If Arg4 and/or Arg5 are zero or a negative value, the command string will not contain position information like the example below: `CALIB,viscal1`

Example return strings from MVS (calibration status value):

```
20
```

It is up to the teach pendant program to correctly interpret the calibration return status from the MVS.

1.3 MVS_SET_PROJECT

This KAREL program will send a command to set the current vision project to the MVS. Some third-party vision systems may have the same name vision process in multiple projects, and the currently selected project dictates which vision task is actually run.

Refer to Part II, Section 2.4 for an explanation of arguments.

1.3.1 MVS_SET_PROJECT Command and Return String Example

The FANUC system will send a project name to set.

Example string from FANUC:

```
SET_PROJECT,GET_PART9
```

The MVS shall return a numeric status value. It is up to the teach pendant program to correctly interpret the return status value.

Examples of return strings from MVS:

```
01
```

1.4 MVS_INSPECT

This KAREL program is used to send a command to perform an inspection operation with the external MVS.

Refer to Part II, Section 2.8 for an explanation of arguments.

1.4.1 MVS_INSPECT Example Command String and Return Strings

The FANUC robot will send a command `INSPECT` followed by a string (max length 32 characters) denoting the vision system job to run.

Example string from FANUC:

```
INSPECT,PART3
```

The MVS will respond with `PASS` or `FAILURE` or `UNKNOWN`. For `UNKNOWN` a string of up to 246 characters can follow providing a reason for indeterminate result.

Example return strings from MVS:

```
PASSFAILUREUNKNOWN,GenericVis-9996 Inspection process not defined012
```

1.5 MVS_READ_BARCODE

This KAREL program will send a command to perform a read operation with the external MVS.

Refer to Part II, Section 2.7 for an explanation of arguments.

1.5.1 MVS_READ_BARCODE Example Command String and Return Strings

The FANUC robot will send a command `READ_BARCODE` followed by a string (max length 32 characters) denoting the vision system job to run.

Example string from FANUC:

```
READ_BARCODE,CODE2
```

The MVS will respond with `SUCCESS` or `0`, or `FAILURE` or non-zero integer value. For `SUCCESS` (or `0`) the following data will follow:

A string up to 246 characters that were read by the MVS.

For `FAILURE`, a string of up to 246 characters can follow providing a reason for failure.

Example MVS return strings:

```
SUCCESS,barcode value
FAILURE,GenericVis-9995 Reading failed, no barcode detected
0,barcode value
1
1,error message text
```

1.6 MVS_SEND_CMD

This function is provided as a flexible solution for future commands which may not fit into the existing API commands. The TP program will have to generate and parse the strings sent/received with this command. The KAREL program simply sends the command to the MVS, and populates a string register and numeric status register. If you wish to send strings longer than 34 characters, see `MVS_SEND_SR`.

Refer to Part II, Section 2.9 for an explanation of arguments.

Example calls:

```
CALL MVS_SEND_CMD(1, SR[1], 1, 2)
```

```
CALL MVS_SEND_CMD(1, any text up to 34 characters here, 1, 2)
```

Example string from FANUC: `any text up to 34 characters here.`

Example return string from MVS: `1,any text which does not exceed a total string length of 254 characters here.`

The entire string returned from the MVS will be stored in the SR[] index specified in Arg4.

The register index for Arg3 will contain any error status returned by the read/write functions used to communicate with the MVS. It is up to the TP program to use `MVS_PARSE` to correctly parse and evaluate the return data in the SR[] index.

1.7 MVS_SEND_SR

This KAREL program is provided as a flexible solution for future commands which may not fit into the existing API commands. The TP program will have to generate and parse the strings sent/received with this command. The KAREL program simply sends the command which is stored in a string register (SR[]) to the MVS, then populates the SR[] return string with the string received from the MVS.

Refer to Part II, Section 2.10 for an explanation of arguments.

Example calls:

```
CALL MVS_SEND_SR(1, 1, 1, 2)
```

Example string from FANUC: `any text up to 254 characters`

Example return string from MVS: `1,any text which does not exceed a total string length of 254 characters here`

The entire string returned from the MVS will be stored in the SR[] index specified in Arg4.

The register index for Arg3 will contain any error status returned by the read/write functions used to communicate with the MVS. It is up to the TP program to use `MVS_PARSE` to correctly parse and evaluate the return data in the SR[] index.

1.8 MVS_PARSE

This KAREL program takes the value in SR[Arg1] and parses the first comma delineated value, storing it in SR[Arg2]. SR[Arg1] will be the remainder of the string with the first comma-delineated value removed from the string. This program is intended to be used with MVS_SEND_CMD to allow the TP program to parse comma-delineated strings more easily.

Example:

If the contents of SR[1] = ’22,text,37’ and SR[2] = ‘’ , then the TP program calls MVS_PARSE:

```
CALL MVS_PARSE(1,2)
```

The resulting values of SR[1] and SR[2] after calling this command will be:

SR[1] = ‘text,37’

SR[2] = ‘22’

Here is an example of using MVS_SEND_CMD and MVS_PARSE to put result values into a position register.

```
1: CALL MVS_SEND_CMD(1, “MEASURE”,1,1) ;
2: ! Check for error ;
3: IF R[1]<>0 JMP,LBL[999] ;
4: ! Get status from the MVS
5: CALL MVS_PARSE(1,2) ;
6: ! Check the first value, if OK proceed ;
7: IF SR[2]<>”OK” JMP,LBL[999] ;
8: !Get X, then Y, then R values, store in SR[3], SR[4], SR[5]
9: CALL MVS_PARSE(1,3) ;
10: CALL MVS_PARSE(1,4) ;
11: CALL MVS_PARSE(1,5) ;
12: PR[1,1]=SR[3] ;
13: PR[1,2]=SR[4] ;
14: PR[1,3]=0 ;
15: PR[1,4]=0 ;
16: PR[1,5]=0 ;
17: PR[1,6]=SR[5] ;
```

PART IV
USING ROBOGUIDE TO TEST EVI INTERFACE

Topics:

1 USING ROBOGUIDE TO TEST THE EVI INTERFACE

ROBOGUIDE is a PC product that provided a 3D virtual environment for simulating a FANUC robot cell. It runs just like the real controller. The EVI runs the same on ROBOGUIDE as it does on a real robot. Thus, ROBOGUIDE can effectively be used to debug the external vision system using the EVI interface.

1.1 ROBOGUIDE OVERVIEW

ROBOGUIDE simulation and its family of process plug-ins provides an easy to use interface to create workcells and robot programs. ROBOGUIDE simulation provides offline process programming for FANUC Robots. ROBOGUIDE simulation is centered on an offline 3D-World, and includes robot workcell modeling, TPP Program teaching, and path playback. Accurate cycle time and robot motion trajectory data are outputs from the ROBOGUIDE simulation teaching system. ROBOGUIDE contains the FANUC virtual robot, which is the actual robot controller software running on a PC.

ROBOGUIDE will provide very accurate robot motion cycle times. However, other CPU intensive operation will be affected by the processing capabilities of the PC that ROBOGUIDE is executing on. Thus, some timing differences may be observed for some functionality; such as communication timing.

Note

The following options are needed for EVI to work properly in ROBOGUIDE:

- Ext Mach Vis I/F Opt (R897)
- User Socket Msg (R648)

1.2 HOST COMM SETUP

The IP address used by the virtual robot is not affected by the TCP/IP setup screen of the Host Comm setup. It will use the ip address of the PC it is running on. Alternatively, on the PC you're running Roboguide on, you can also access the robot from the loop back address of `localhost` or `127.0.0.1`.

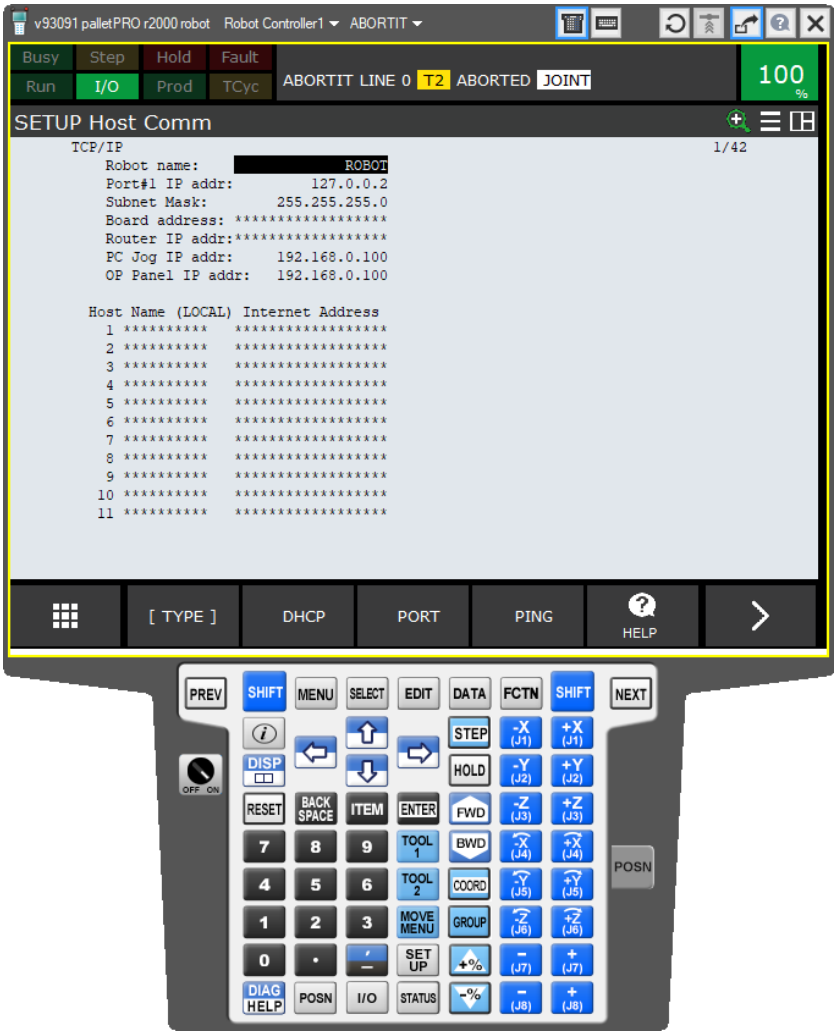
Refer to Part II, Section 1.2, SETTING UP A CLIENT TAG for how to configure the client tag for communicating with the external vision system.

If you have more than one Ethernet interface on your PC, ROBOGUIDE will bind to the first interface it finds. If you want to use a specific interface, you may need to temporarily disable the other interfaces while ROBOGUIDE starts up.

There is a file called services.txt which lists the port numbers and loop back addresses that ROBOGUIDE uses for connections. This file can be found under the directory for the robot under the ROBOGUIDE Project folder. There is a shortcut to the ROBOGUIDE project folder under the Tools Menu; the first item is **RG Project Name** Folder.

Note

The remainder of the settings for EVI are identical to those on the real robot.



APPENDIX

A FANUC 3D ORIENTATION CONVENTION

FANUC uses the $Z_1Y_2X_3$ Euler angle convention shown in the table below. Orientation angles need to be in the range +/- 180°. Any angle over 180 degrees needs to have 360° subtracted from it to put it into this range.

Proper Euler angles		Tait-Bryan angles	
$X_1Z_2X_3 = \begin{bmatrix} c_2 & -c_3s_2 & s_2s_3 \\ c_1s_2 & c_1c_2c_3 - s_1s_3 & -c_3s_1 - c_1c_2s_3 \\ s_1s_2 & c_1s_3 + c_2c_3s_1 & c_1c_3 - c_2s_1s_3 \end{bmatrix}$		$X_1Z_2Y_3 = \begin{bmatrix} c_2c_3 & -s_2 & c_2s_3 \\ s_1s_3 + c_1c_3s_2 & c_1c_2 & c_1s_2s_3 - c_3s_1 \\ c_3s_1s_2 - c_1s_3 & c_2s_1 & c_1c_3 + s_1s_2s_3 \end{bmatrix}$	
$X_1Y_2X_3 = \begin{bmatrix} c_2 & s_2s_3 & c_3s_2 \\ s_1s_2 & c_1c_3 - c_2s_1s_3 & -c_1s_3 - c_2c_3s_1 \\ -c_1s_2 & c_3s_1 + c_1c_2s_3 & c_1c_2c_3 - s_1s_3 \end{bmatrix}$		$X_1Y_2Z_3 = \begin{bmatrix} c_2c_3 & -c_2s_3 & s_2 \\ c_1s_3 + c_3s_1s_2 & c_1c_3 - s_1s_2s_3 & -c_2s_1 \\ s_1s_3 - c_1c_3s_2 & c_3s_1 + c_1s_2s_3 & c_1c_2 \end{bmatrix}$	
$Y_1X_2Y_3 = \begin{bmatrix} c_1c_3 - c_2s_1s_3 & s_1s_2 & c_1s_3 + c_2c_3s_1 \\ s_2s_3 & c_2 & -c_3s_2 \\ -c_3s_1 - c_1c_2s_3 & c_1s_2 & c_1c_2c_3 - s_1s_3 \end{bmatrix}$		$Y_1X_2Z_3 = \begin{bmatrix} c_1c_3 + s_1s_2s_3 & c_3s_1s_2 - c_1s_3 & c_2s_1 \\ c_2s_3 & c_2c_3 & -s_2 \\ c_1s_2s_3 - c_3s_1 & c_1c_3s_2 + s_1s_3 & c_1c_2 \end{bmatrix}$	
$Y_1Z_2Y_3 = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_1s_2 & c_3s_1 + c_1c_2s_3 \\ c_3s_2 & c_2 & s_2s_3 \\ -c_1s_3 - c_2c_3s_1 & s_1s_2 & c_1c_3 - c_2s_1s_3 \end{bmatrix}$		$Y_1Z_2X_3 = \begin{bmatrix} c_1c_2 & s_1s_3 - c_1c_3s_2 & c_3s_1 + c_1s_2s_3 \\ s_2 & c_2c_3 & -c_2s_3 \\ -c_2s_1 & c_1s_3 + c_3s_1s_2 & c_1c_3 - s_1s_2s_3 \end{bmatrix}$	
$Z_1Y_2Z_3 = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_3s_1 - c_1c_2s_3 & c_1s_2 \\ c_1s_3 + c_2c_3s_1 & c_1c_3 - c_2s_1s_3 & s_1s_2 \\ -c_3s_2 & s_2s_3 & c_2 \end{bmatrix}$		$Z_1Y_2X_3 = \begin{bmatrix} c_1c_2 & c_1s_2s_3 - c_3s_1 & s_1s_3 + c_1c_3s_2 \\ c_2s_1 & c_1c_3 + s_1s_2s_3 & c_3s_1s_2 - c_1s_3 \\ -s_2 & c_2s_3 & c_2c_3 \end{bmatrix}$	
$Z_1X_2Z_3 = \begin{bmatrix} c_1c_3 - c_2s_1s_3 & -c_1s_3 - c_2c_3s_1 & s_1s_2 \\ c_3s_1 + c_1c_2s_3 & c_1c_2c_3 - s_1s_3 & -c_1s_2 \\ s_2s_3 & c_3s_2 & c_2 \end{bmatrix}$		$Z_1X_2Y_3 = \begin{bmatrix} c_1c_3 - s_1s_2s_3 & -c_2s_1 & c_1s_3 + c_3s_1s_2 \\ c_3s_1 + c_1s_2s_3 & c_1c_2 & s_1s_3 - c_1c_3s_2 \\ -c_2s_3 & s_2 & c_2c_3 \end{bmatrix}$	

INDEX

REVISION RECORD

Edition	Date	Contents
02	May., 2022	● A note about stack size is added.
01	Nov., 2020	● First release.

B-84244EN/02